Introduction to PIC Programming

Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 4: Using Timer0

The lessons until now have covered the essentials of midrange PIC microcontroller operation: controlling digital outputs, timed via programmed delays, with program flow responding to digital inputs. That's enough to allow you to perform a great many tasks. 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 <u>lesson 1</u>, to accurately measure elapsed time. But a timer will keep counting, steadily, while your program responds to inputs, performs calculations, or whatever.

As we'll see in <u>lesson 6</u>, timers are commonly used to drive *interrupts* (routines which interrupt the normal program flow) to allow regularly timed "background" tasks to run. However, before moving on to timer-based interrupts, it's important to understand how timers operate. And, as this lesson will demonstrate, timers can be very useful, even when not used with interrupts.

This lesson revisits the material in <u>baseline lesson 5</u>, covering:

- Introduction to the Timer0 module
- Creating delays with Timer0
- Debouncing via Timer0
- Using Timer0 counter mode with an external clock (demonstrating the use of a crystal oscillator as a time reference)

Timer0 Module

Midrange PICs can have up to three timers; the simplest of these is referred to as Timer0. 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, sets an "overflow flag" (the TOIF bit in the INTCON register, triggering an interrupt if Timer0 interrupts are enabled) to indicate that the rollover happened, and then continues to increment.

Note that this is different from the Timer0 module in the baseline architecture, which does not have an overflow flag.

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_REG	GPPU	INTEDG	TOCS	TOSE	PSA	PS2	PS1	PS0

The clock source is selected by the **TOCS** bit:

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

TOCS = 1 selects counter mode, where TMR0 is incremented by an external signal, on the TOCKI pin. On the PIC12F629, this is physically the same pin as GP2.

TOCKI 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 TOSE bit selects whether Timer0 responds to rising or falling signals ("edges") on TOCKI. Clearing TOSE to '0' selects the rising edge; setting TOSE to '1' selects the falling edge.

Prescaler

By default, the timer increments by one for every instruction cycle (in timer mode) or transition on TOCKI (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, TMRO will increment every 1 μ s. Thus, with a 4 MHz clock, the maximum period that TimerO can measure directly, by default, is 255 μ s.

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^{1} .

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 $255 \times 256 \ \mu s = 65.28 ms$.

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

If you don't want to use the prescaler with Timer0, for a 1:1 "prescale ratio", set PSA to '1'.

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

¹ If PSA = 1, the prescaler is assigned to the watchdog timer – a topic covered in <u>lesson 7</u>.

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 a LED to the circuit used in lesson 3. A second LED has been added to GP2, although any of the unused pins would have been suitable.

You may wish to make the two LEDs different colours, for example red on GP1 and green on GP2.



To illustrate how Timer0 can be used to measure elapsed time, we'll implement a very simple reaction time "game": light a LED to indicate 'start', and then 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. Then reset and repeat.

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
      clear 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 μ s. Using the prescaler, with a ratio of 1:32, means that the timer increments every 32 μ s. If we clear TMR0 and then wait until TMR0 = 250, 8 ms $(250 \times 32 \,\mu s)$ will have elapsed. If we then reset TMR0 and increment a counter variable, we've implemented a counter which increments every 8 ms. Since



 $25 \times 8 \text{ ms} = 200 \text{ ms}$, when the counter reaches 25, 200 ms will have elapsed; any counter value > 25 means that the allowed time has been exceeded. And since $125 \times 8 \text{ ms} = 1 \text{ s}$, when the counter reaches 125, 1 s will have elapsed and we can stop waiting for the button press.

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

```
movlw b'11000100' ; configure Timer0:
 ; --0----- ; -----0--- ; configure Timer0 (TOCS = 0)
 ; -----100 prescaler assigned to Timer0 (PSA = 0)
 prescale = 32 (PS = 100)
banksel OPTION_REG ; -> increment TMR0 every 32us
```

This code is setting bits 6 and 7 of OPTION_REG, even though these bits (GPPU and INTEDG) are not related to Timer0. In the baseline architecture, there is no choice but to load the whole of the OPTION register at once, but for midrange PICs it is possible to use bit set/clear instructions to modify individual bits in OPTION_REG, or to use logical *masking* operations to update only some bit fields, leaving other bits unchanged. For example, to preserve the contents of OPTION_REG<6:7>, you could write:

banksel	OPTION_REG		
movf	OPTION_REG,w	; operate on OPTION_REG	
andlw	b'11000000'	; while preserving bits 6-7	
iorlw	b'00000100'		
	;0	timer mode (TOCS = 0)	
	;0	prescaler assigned to Timer0 (PSA :	= 0)
	;100	prescale = 32 (PS = 100)	
movwf	OPTION_REG	; -> increment TMR0 every 32us	

The 'andlw' and 'iorlw' instructions respectively perform "logical and" and "inclusive-or" operations on the W register with the given literal (constant) value, placing the result in W – "and literal with W" and "inclusive-or literal with W".

However, given that, by default (after a power-on reset), every bit in OPTION_REG is set to '1', there is no real need to go to the trouble to use masks to preserve bits 6 and 7; we know that they were already set to '1'. Nevertheless, in some cases you will want to update only part of a register, so it's worth taking the time to understand how these masking operations work. There will be more examples in later lessons.

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

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

```
clrf TMR0
w_tmr0 movf TMR0,w ; wait for 8ms (250x32us)
xorlw .250
btfss STATUS,Z
goto w_tmr0
```

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 1 and 2.

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.

However long the additional code is, it takes some time to run, so the timer increment will not be detected immediately. This means that the overall delay will be a little longer than intended. For that reason (and others), it is usually better to use timer-driven interrupts for tasks like this, as we will see in lesson 6.

That's not a problem in this example, where exact timing is not important, so 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.

For example:

```
w tmr0
       banksel GPIO
       btfss
               GPIO,GP3
                             ; check for button press (GP3 low)
        goto
               btn dn
       banksel TMR0
       movf
               TMR0,w
       xorlw
                .250
                                ; wait for 8ms (250x32us)
               STATUS,Z
       btfss
       goto
               w_tmr0
```

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:

```
clrf
                cnt8ms
                                ; clear 8ms counter
wait1s
       banksel TMR0
                                ; clear timer0
       clrf
               TMR0
w tmr0
       banksel GPIO
       btfss GPIO,GP3
                               ; check for button press (GP3 low)
        goto
               btn dn
       banksel TMR0
       movf
               TMR0,w
       xorlw
               .250
                               ; wait for 8ms (250x32us)
               STATUS,Z
       btfss
               w_tmr0
       goto
              cnt8ms,f
                               ; increment 8ms counter
        incf
               .125
                               ; continue to wait for 1s (125x8ms)
       movlw
       xorwf
               cnt8ms,w
       btfss STATUS,Z
       goto
               wait1s
```

The test at the end of the outer loop (cnt8ms = 125) ensures that the loop completes when 1 s has elapsed, in case the button has not 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 $A \ge B$, A - B is positive or zero and no borrow is needed.

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

Midrange PICs have two subtraction instructions:

'subwf f, d'-"subtract W from file register", where 'f' is the register and, 'd' is the destination; ', f' to write the result back to the register: f = f - W', w' to place the result in W: W = f - W

and:

'sublw k' - "subtract **W** from literal", where 'k' is the literal value to subtract W from; the result is placed in W: W = k - W Note that there is no instruction which subtracts a literal from W. Or is there?

Recall that the expression 'W – k' is equivalent to 'W + (–k)', i.e. adding a negative value is equivalent to subtracting a positive value.

We saw in <u>baseline lesson 10</u> that when negative values are represented in *two's complement* format, the normal binary integer addition and subtraction operations continue to work, in a consistent way, with both positive and negative numbers.

The 'addlw' instruction is used to add a literal to W.

The '-' operator is used by the MPASM assembler to specify a two's complement value, so to subtract a literal from W, we can simply write:

'addlw -k', which performs the operation: W = W - k

Whichever way the subtraction is performed, the result 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	IRP	RP1	RP0	ТО	PD	Z	DC	С

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	С
A > B	0	1
A = B	1	1
A < B	0	0

We can make use of this to test whether the elapsed time is less than 200 ms (cnt8ms < 25) as follows:

movlw	.25	;	if	time	e <	200ms	(25	Х	8ms)
subwf	cnt8ms,w	;	(cr	nt8ms	s <	25)			
banksel	GPIO								
btfss	STATUS,C								
bsf	GPIO,GP1	;	t	curn	on	success	s L	ΕD	

The subtraction performed here is cnt8ms - 25, so C = 0 only if cnt8ms < 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.

Note that the 'banksel GPIO' directive is placed above the 'btfss' instruction. This is important. If we had instead written this as:

```
btfss STATUS,C
banksel GPIO
bsf GPIO,GP1 ; turn on success LED
```

the instruction generated by $banksel^2$ is skipped if C is set, instead of the bcf instruction.

² On midrange PICs with four register banks, banksel generates two instructions; only the first will be skipped.

That is not at all what was intended; keep in mind that the 'banksel' directive generates instructions which are inserted into your code, so sometimes (as in this example) you need to be careful where you place it, to avoid unexpected side-effects.

Note also that there is never any need to use banksel before accessing the STATUS register, because it is mapped at the same address in every bank.

Alternatively, we could use the sublw instruction to perform the comparison:

```
btn_dn movf cnt8ms,w ; if time < 200ms (25 x 8ms)
sublw .24 ; (cnt8ms <= 24)
banksel GPIO
btfsc STATUS,C
bsf GPIO,GP1 ; turn on success LED</pre>
```

Note that the sense of the subtraction performed here (24 - cnt8ms) is reversed from the one above. According to the truth table on the previous page, we now have to test for C = 1 instead of C = 0 and the comparison becomes ' \leq ' instead of '<', meaning that the comparison has to be with 24 instead of 25.

Or, we could even use addlw for the subtraction (comparison):

```
movf cnt8ms,w ; if time < 200ms (25 x 8ms)
addlw -.25 ; (cnt8ms < 25)
banksel GPIO
btfss STATUS,C
bsf GPIO,GP1 ; turn on success LED</pre>
```

Complete program

Here's the complete code for the reaction timer, using the 'subwf'-based comparison routine:

```
*****
                                                       *
   Description:
               Lesson 4, example 1a
;
                                                       *
               Reaction Timer game.
;
                                                       *
;
   Demonstrates use of timer0 to time real-world events
;
;
   User must attempt to press button within 200ms 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
;
;
   (version using subwf instruction in comparison routine)
;
;
*
;
                                                       *
   Pin assignments:
;
     GP1 - success LED
                                                       *
;
                                                       *
;
      GP2 - start LED
                                                       *
      GP3 - pushbutton
;
;
p=12F629
   list
```

#include <p12F629.inc>

errorlevel -302 ; no "register not in bank 0" warnings errorlevel -312 ; no "page or bank selection not needer ; no "page or bank selection not needed" messages ;***** CONFIGURATION ; int reset, no code or data protect, no brownout detect, ; no watchdog, power-up timer, 4Mhz int clock MCLRE OFF & CP OFF & CPD OFF & BODEN OFF & WDT OFF & CONFIG PWRTE ON & INTRC OSC NOCLKOUT ;***** EXTERNAL LABELS EXTERN delay10 ; W x 10ms delay ;***** VARIABLE DEFINITIONS UDATA SHR cnt8ms res 1 ; 8ms counter (incremented every 8ms) RESET CODE 0x0000 ; processor reset vector ; calibrate internal RC oscillator call 0x03FF ; retrieve factory calibration value banksel OSCCAL ; then update OSCCAL movwf OSCCAL ;***** Initialisation ; configure port movlw ~(1<<GP1|1<<GP2) ; configure GP1 and GP2 (only) as outputs banksel TRISIO movwf TRISIO ; configure timer movlw b'11000100' ; configure Timer0: ; ------- timer mode (TOCS = 0) ; -----100 prescaler assigned to Timer0 (PSA = 0) prescale = 32 (PS = 100) banksel OPTION_REG ; -> increment TMR0 every 32us movwf OPTION REG ;***** Main loop ; start with all LEDs off loop banksel GPIO clrf GPIO ; delay 2s movlw .200 pagesel delay10 call delay10 ; 200 x 10 ms delay pagesel \$; turn on start LED banksel GPIO bsf GPIO,GP2 ; wait for button press clrf cnt8ms ; clear 8ms counter wait1s banksel TMR0 ; clear timer0 clrf TMR0 w tmr0 banksel GPIO btfss GPIO,GP3 ; check for button press (GP3 low) goto btn_dn

```
banksel TMR0
       movf
               TMR0,w
       xorlw
               .250
                              ; wait for 8ms (250 x 32us)
       btfss STATUS,Z
       JULO W_tmr0
incf crt^
              cnt8ms,f
                              ; increment 8ms counter
       movlw
               .125
                               ; continue to wait for 1s (125 x 8ms)
       xorwf
               cnt8ms,w
       btfss STATUS,Z
       goto
               wait1s
       ; check elapsed time
btn_dn movlw .25
subwf cnt8ms,w
                              ; if time < 200ms (25 x 8ms)
                              ; (cnt8ms < 25)
       banksel GPIO
       btfss STATUS,C
       bsf
              GPIO,GP1
                              ; turn on success LED
        ; delay 1s
       movlw .100
       pagesel delay10
                             ; 100 x 10 ms delay
       call delay10
       pagesel $
        ; repeat forever
              loop
       goto
       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 the second example in lesson 1), while lighting the LED on GP1 whenever the pushbutton on GP3 is pressed (as was done in lesson 3).

We'll see in <u>lesson 6</u> that timer-driven interrupts are ideally suited to performing regular background tasks. This example is only included here for completeness; it's not how you would implement this, on a midrange PIC, in practice.

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 <u>lesson 2</u>, 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 s delay is very noticeable).

Since the maximum delay that Timer0 can produce directly from a 1 MHz instruction clock is 65 ms, we have to extend the timer by adding a counter variable, as was done 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 $125 \times 125 \times 32 \ \mu s = 500 \ ms$, a delay of exactly 500 ms can be generated by:

- Using a 4 MHz processor clock, giving 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 $125 \times 32 \ \mu s = 4 \ ms$)
- Repeating 125 times, creating a delay of 125×4 ms = 500 ms.

The following code implements the above steps:

```
;***** Initialisation
                         ; configure port
                         movlw ~ (1<<GP1|1<<GP2) ; configure GP1 and GP2 as outputs
herbed [JEG10] ; (CD2 is an input)</pre>
                         banksel TRISIO
                                                                                                                   ; (GP3 is an input)
                         movwf TRISIO
                         ; configure timer
                         movlw b'11000100' ; configure Timer0:
                         indexist is integrated in the integrated is integrated in the integration is integrated in the in
                         movwf OPTION REG
                          ; initialise port
                         banksel GPIO
                         clrf GPIO ; start with all LEDs off
clrf sGPIO ; update shadow
;**** Main loop
                        ; delay 500ms
loop
                                                                                                       ; repeat 125 times
                        movlw .125 ; repeat 125 times
movwf dlycnt ; (125 x 4ms = 500ms)
                         movlw .125
dly500 ; (begin 500ms delay loop)
                         banksel TMR0
                                                                                                       ; clear timer0
                         clrf
                                                TMR0
                           ; check timer0 until 4ms elapsed
w tmr0
                         movf
                                              TMRO,w
                         xorlw
                                                   .125
                                                                                                     ; (4ms = 125 \times 32us)
                         btfss STATUS,Z
                         goto w tmr0
                          ; (end 500ms delay loop)
                         decfsz dlycnt,f
goto dly500
                          ; toggle LED
                         movf sGPIO,w
xorlw 1<<GP2
                                                                                                ; toggle LED on GP2
; using shadow register
                         movwf sGPIO
                         banksel GPIO
                         movwf GPIO
                          ; repeat forever
                          goto loop
```

Note that, strictly speaking, the 'banksel' directives within the main loop are not needed, because the only registers accessed within the loop, TMRO and GPIO, are in the same bank. Nevertheless, it's good practice to include these directives, as shown, in case you later insert some code which changes the bank selection. That's not difficult to deal with, but it's easy to miss a situation where banksel is needed, ending up with a difficult-to-find bug. If you use banksel liberally, even when not strictly needed, your code will be a little longer, but much more easily maintained.

Here's the code developed in lesson 3, for turning on a LED when the pushbutton is pressed:

clrf	sGPIO	;	assume button up -> LED off
btfss bsf	GPIO,GP3 sGPIO,GP1	; ;	if button pressed (GP3 low) turn on LED
movf movwf	sGPIO,w GPIO	;	copy shadow to 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:
```

```
; check and respond to button press
w tmr0
       banksel GPIO
                           ; assume button up -> LED off
       bcf
             sGPIO,GP1
       btfss GPIO,GP3
                            ; if button pressed (GP3 low)
             sGPIO,GP1
                            ; turn on LED
       bsf
       movf sGPIO,w
                            ; copy shadow to GPIO
       movwf GPIO
          ; check timer0 until 4ms elapsed
       banksel TMR0
       movf TMR0,w
       xorlw
                            ; (4ms = 125 x 32us)
              .125
       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, 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 loop.

```
Description: Lesson 4 example 2
                                                       *
;
;
  Demonstrates use of TimerO to maintain timing of background tasks
;
  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
;
list
           p=12F629
   #include <p12F629.inc>
  errorlevel -302
                       ; no "register not in bank 0" warnings
;***** CONFIGURATION
            ; int reset, no code or data protect, no brownout detect,
            ; no watchdog, power-up timer, 4Mhz int clock
            MCLRE OFF & CP OFF & CPD OFF & BODEN OFF & WDT OFF &
    CONFIG
PWRTE ON & INTRC OSC NOCLKOUT
;***** VARIABLE DEFINITIONS
     UDATA SHR
```

sGPIO res 1 ; shadow copy of GPIO dlycnt res 1 ; delay counter RESET CODE 0x0000 ; processor reset vector ; calibrate internal RC oscillator call 0x03FF ; retrieve factory calibration value banksel OSCCAL ; then update OSCCAL movwf OSCCAL ;***** Initialisation ; configure port movlw ~(1<<GP1|1<<GP2) ; configure GP1 and GP2 as outputs banksel TRISIO ; (GP3 is an input) movwf TRISIO ; configure timer ; configure timer movlw b'11000100' ; configure Timer0: ; ------ timer mode (TOCS = 0) ; -----100 prescaler assigned to Timer0 (PSA = 0) prescale = 32 (PS = 100) banksel OPTION_REG ; -> increment TMR0 every 32 us movwf OPTION REG ; initialise port banksel GPIO clrf GPIO ; start with all LEDs off clrf sGPIO ; update shadow ;**** Main loop ; delay 500ms loop movlw .125 ; repeat 125 times
movwf dlycnt ; (125 x 4ms = 500ms) dly500 ; (begin 500ms delay loop) banksel TMR0 ; clear timer0 clrf TMR0 ; check and respond to button press w tmr0 banksel GPIO bcfsGPIO,GP1; assume button up -> LED offbtfssGPIO,GP3; if button pressed (GP3 low)bsfsGPIO,GP1; turn on LED movf sGPIO,w ; copy shadow to GPIO movwf GPIO ; check timer0 until 4ms elapsed banksel TMR0 movf TMR0,w xorlw .125 ; $(4ms = 125 \times 32us)$ btfss STATUS,Z goto w tmr0 ; (end 500ms delay loop) decfsz dlycnt,f goto dly500 ; toggle LED movf sGPIO,w xorlw 1<<GP2 ; toggle LED on GP2 movwf sGPIO ; using shadow register ; repeat forever goto loop END

Example 3: Switch debouncing

<u>Lesson 3</u> 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</pre>
```

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 determined by continuing to increment a count while checking the state of the switch. "Continuing to increment a count" while something else occurs, such as checking a switch, 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</pre>
```

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 \div 64 μ s = 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,GP3 ; 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 3, and it avoids the need to use two data registers as counters. But – it uses Timer0. Although midrange PICs have more than one timer, they are still a scarce resource. You must be careful, as you build a library of routines that use Timer0, that if you use more than one routine which uses Timer0 in a single program, that the way they use or setup Timer0 doesn't clash. As we'll see in lesson 6, it can be better to use a regular timer-driven interrupt for switch debouncing, allowing a single timer (driving the interrupt) to be used for a number of tasks.

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 <u>lesson 3</u>:

* ; Pin assignments: ; GP1 - indicator LED ; GP3 - pushbutton ; ; p=12F629 list #include <p12F629.inc> errorlevel -302 ; no "register not in bank 0" warnings ;***** CONFIGURATION ; int reset, no code or data protect, no brownout detect, ; no watchdog, power-up timer, 4Mhz int clock CONFIG MCLRE OFF & CP OFF & CPD OFF & BODEN OFF & WDT OFF & PWRTE ON & INTRC OSC NOCLKOUT ;***** VARIABLE DEFINITIONS UDATA_SHR sGPIO res 1 ; shadow copy of GPIO RESET CODE 0x0000 ; processor reset vector ; calibrate internal RC oscillator call 0x03FF ; retrieve factory calibration value banksel OSCCAL ; then update OSCCAL movwf OSCCAL ;***** Initialisation ; configure port movlw ~ (1<<GP1) ; configure GP1 (only) as an output banksel TRISIO ; (GP3 is input only) movwf TRISIO movwf TRISIO ; configure timer movlw b'11000101' ; configure Timer0: ; --0----- timer mode (TOCS = 0) ; ----0--- prescaler assigned to Timer0 (PSA = 0) ; -----U--- prescaler assigned to Timer0 ; -----101 prescale = 64 (PS = 101) banksel OPTION_REG ; -> increment TMR0 every 64 us movwf OPTION REG ; initialise port banksel GPIO clrf GPIO ; start with all LEDs off clrf sGPIO ; update shadow ;**** Main loop loop ; wait until button pressed, debounce using timer0: wait_dn clrf TMR0 ; reset timer chk_dn btfsc GPIO,GP3 ; 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
        banksel GPIO
        movf
                sGPIO,w
        xorlw
                1<<GP1
                                 ; toggle shadow register
        movwf
                sGPIO
        movwf
                GPIO
                                 ; write to port
        ; wait until button released, debounce using timer0:
        banksel TMR0
wait up clrf
                TMR0
                                 ; reset timer
chk_up
       btfss
                GPIO, GP3
                                 ; 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
        qoto
                chk up
        ; repeat forever
                loop
        goto
        END
```

Counter Mode

As mentioned above, Timer0 can also be used to count transitions (rising or falling) on the TOCKI input.

This is useful in a number of ways, such as performing an action after some number of events, 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 one second.

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

To illustrate the use of 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 shown on the right.

An oscillator based on a 32.768 kHz "watch crystal" and a CMOS inverter was presented in <u>baseline lesson 5</u>. It is used again here to generate a 32.768 kHz clock signal, which drives the 12F629's TOCKI input, via an inverting buffer.

If you build the 32.768 kHz clock module on a prototype board and are using the Low Pin Count Demo Board, the output of the buffer can be connected to TOCKI via the 14-pin header on the demo board (GP2 is brought out as pin 9 on the header,



while power and ground, which can be used to power the clock module, are on pins 13 and 14).

We'll use this clock input 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 TOCKI, TMRO will be incremented 256 times per second. The most significant bit of TMRO (TMRO<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 TOCKI, 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 TOCKI) with a 1:128 prescale ratio, set the TOCS bit to '1', PSA to '0' and PS<2:0> to '110':

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

Note that the value of TOSE bit is irrelevant; we don't care if the counter increments on the rising or falling edge of the signal on TOCKI – 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 midrange PIC assembler!

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

```
loop ; transfer TMR0<7> to GP1
banksel TMR0
btfsc TMR0,7 ; if TMR0<7>=1
bsf GPIO,GP1 ; set GP1
btfss TMR0,7 ; if TMR0<7>=0
bcf GPIO,GP1 ; clear GP1
; repeat forever
goto loop
```

As we saw in <u>lesson 3</u>, if you are using a shadow register (generally a good idea), this can be implemented as:

```
loop ; transfer TMR0<7> to GP1
clrf sGPIO ; assume TMR0<7>=0 -> LED off
banksel TMR0
btfsc TMR0,7 ; if TMR0<7>=1
bsf sGPIO,GP1 ; turn on LED
movf sGPIO,w ; copy shadow to GPIO
banksel 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.

And note that the use of a single 'banksel' directive at the start of the first routine, but two 'banksel's in the second. This is because, in a real program, where a shadow register is being used, it is likely to be updated a number of times before being copied to GPIO at the end of the loop; additional code within the loop may alter the bank selection.

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:





'rrf f, d'- "rotate right file 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 to TMR0<7> into C, without affecting the value in TMR0.

In the midrange architecture, only the special-function and general purpose registers can be rotated; there are no instructions for rotating W. That's a pity, since such an instruction would be useful here.

Instead, we must rotate the bit copied from TMR0<7> into bit 0 of a temporary register, then another rotate to move the copied bit into bit 1, and then copy the result to GPIO, as follows:

rlf	TMRO,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 = $TMRO < 7>$)

Note that 'temp' is cleared before being used. That's not strictly necessary in this example; since the only output is GP1, it doesn't 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, 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 an LED at 1 Hz using a crystal oscillator" program, using the "copy a bit via rotation" method:

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

; * Pin assignments: ; * GP1 - flashing LED ; * TOCKI - 32.768 kHz signal ; ; p=12F629 list <p12F629.inc> #include errorlevel -302 ; no "register not in bank 0" warnings ;***** CONFIGURATION ; ext reset, no code or data protect, no brownout detect, ; no watchdog, power-up timer, 4Mhz int clock MCLRE ON & CP OFF & CPD OFF & BODEN OFF & WDT OFF & CONFIG PWRTE ON & INTRC OSC NOCLKOUT ;***** VARIABLE DEFINITIONS UDATA SHR temp res 1 ; temp register used for rotates RESET CODE 0x0000 ; processor reset vector ; calibrate internal RC oscillator call 0x03FF ; retrieve factory calibration value banksel OSCCAL ; then update OSCCAL movwf OSCCAL ;***** Initialisation ; configure port movlw ~(1<<GP1) ; configure GP1 (only) as an output banksel TRISIO ; (GP3 is input only) movwf TRISIO ; configure timer movlw b'11110110' ; configure Timer0: ; --1---- counter mode (T ; ----0--- prescaler assig: ; ----110 prescale = 128 ; --1---- counter mode (TOCS = 1) ; ----0--- prescaler assigned to Timer0 (PSA = 0) ; -----110 prescale = 128 (PS = 110) banksel OPTION_REG ; -> increment at 256 Hz with 32.768 kHz input movwf OPTION REG ;**** Main loop ; TMR0<7> cycles at 1Hz loop ; so continually copy to GP1 banksel TMR0 TMR0,w ; copy TMR0<7> to C rlf temp clrf rlf temp,f rlf temp,w ; rotate C into temp ; rotate once more into W (-> W<1> = TMR0<7>) movwf GPIO ; update GPIO with result (-> GP1 = TMR0<7>) ; repeat forever goto loop

Hopefully the examples in this lesson have given you an idea of the flexibility and usefulness of the Timer0 peripheral. But, as mentioned a few times now, one of the most useful applications of timers is to drive interrupts – easily the most significant enhancement in the midrange architecture, and the topic of lesson 6.

But first, in the <u>next lesson</u> we'll take a quick look at some of the features of the MPASM assembler, which can make your code easier to maintain.