Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 2: Using Timer 0

As we saw in the previous lesson, C can be a viable choice for programming digital I/O operations on midrange (14-bit) PICs, although, as we saw, programs written in C can consume significantly more memory (a limited resource on these tiny MCUs) than equivalent programs written in assembler.

This lesson revisits the material from midrange lesson 4 on the Timer0 module: using it to time events, to maintain the timing of a background task, for switch debouncing, and as a counter.

Selected examples are re-implemented using the "free" C compilers from HI-TECH Software: PICC-Lite and HI-TECH C[1] (in "Lite" mode) introduced in lesson 1, and, as was done in that lesson, the memory usage and code length is compared with that of assembler. We'll also see the C equivalents of some of the assembler features covered in midrange lesson 5, including macros.

In summary, this lesson covers:

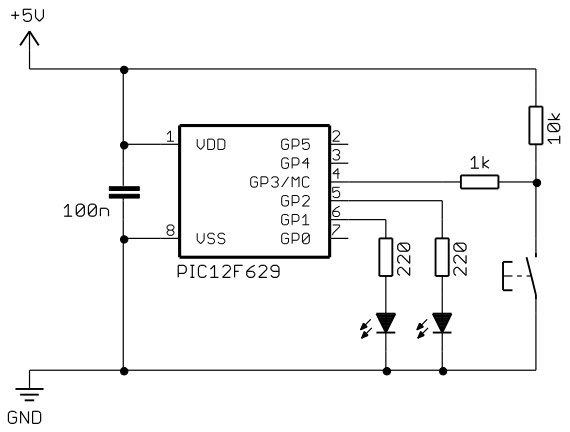
- Configuring Timer0 as a timer or counter
- Accessing Timer0
- Using Timer0 for switch debouncing
- Using C macros

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

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

Example 1: Using Timer0 as an Event Timer

To demonstrate how Timer0 can be used to measure elapsed time, midrange lesson 4 included a "reaction timer" game, using the circuit on the right, where the pushbutton has to be pressed as quickly as possible after the LED on GP2, indicating 'start' is lit. If the button is pressed quickly enough (within a predefined reaction time), the LED on GP1 is lit, to indicate 'success'.



Thus, we need to measure the elapsed time between indicating 'start' and detecting a pushbutton press, and an ideal way to do that is to use Timer0, in its timer mode (clocked by the PIC's instruction clock, which in this example is 1 MHz).

[1] PICC-Lite was bundled with versions of MPLAB up to 8.10. HI-TECH C (earlier known as "HI-TECH C PRO") was bundled with MPLAB 8.15 and later, although you should download the latest version from www.htsoft.com.

Ideally, to make a better "game", the delay before the 'start' LED is lit would be random, but in this simple example, a fixed delay is used.

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
```

To use Timer0 to measure the elapsed time, we need to extend its range (normally limited to 65 ms) by adding a counter variable, which is incremented each time the timer overflows (or reaches a certain value). In the example in midrange lesson 4, Timer0 is configured so that it is clocked every 32 µs, by using the 1 MHz instruction clock with a 1:32 prescaler. After 250 counts, 8 ms (250 × 32 µs) will have elapsed; this is used to increment a counter, which effectively measures time in 8 ms intervals. When the button is pressed, this "8 ms counter" can then be checked, to see whether the maximum reaction time has been exceeded.

As explained in midrange lesson 4, to select timer mode, with a 1:32 prescaler, we must clear the T0CS and PSA bits, in the OPTION register, and set the PS<2:0> bits to 100. This was done by:

```
        movlw   b'11000100'     ; configure Timer0:
                ; --0-----          timer mode (T0CS = 0)
                ; ----0---          prescaler assigned to Timer0 (PSA = 0)
                ; -----100          prescale = 32 (PS = 100)
        banksel OPTION_REG      ;   -> increment TMR0 every 32us
        movwf   OPTION_REG
```

Here is the main assembler code we had used to implement the button press / timing test routine:

```
        ; wait for button press
        clrf    cnt8ms          ; clear 8 ms counter
wait1s
        banksel TMR0            ; clear timer0
        clrf    TMR0
w_tmr0
        banksel GPIO
        btfss   BUTTON          ; check for button press (low)
        goto    btn_dn
        banksel TMR0
        movf    TMR0,w
        xorlw   8000/32         ; wait for 8 ms (32 us/tick)
        btfss   STATUS,Z
        goto    w_tmr0
        incf    cnt8ms,f        ; increment 8 ms counter
        movlw   1000/8          ; continue to wait for 1 s (8 ms/count)
        xorwf   cnt8ms,w
        btfss   STATUS,Z
        goto    wait1s
        ; check elapsed time
btn_dn  movlw   MAXRT/8         ; if time < max reaction time (8ms/count)
        subwf   cnt8ms,w
        banksel GPIO
        btfss   STATUS,C
        bsf     SUCCESS         ;   turn on success LED
```

(This code is actually taken from midrange lesson 5)

HI-TECH C Implementation

As mentioned in the previous lesson, loading the OPTION register in HI-TECH C is done by assigning a value to the variable `OPTION`:

```
OPTION = 0b11000100;            // configure Timer0:
         //--0-----                 timer mode (T0CS = 0)
         //----0---                 prescaler assigned to Timer0 (PSA = 0)
         //-----100                 prescale = 32 (PS = 100)
         //                         -> increment every 32 us
```

Note that this has been commented in a way which documents which bits affect each setting, with '-'s indicating "don't care". For example, we could have instead used '`OPTION = 0b11010100`', since the value of bit 4, or T0SE, is irrelevant in timer mode.

However, some purists would argue, for both assembler and C, that we should be using the symbols defined in the include, or *header* files, instead of binary constants, to make your code easier to understand, more maintainable, and less error-prone (because it is easy to mistype a numeric constant, and the assembler cannot warn you if that happens).

Since the HI-TECH C compilers make the individual OPTION register bits available as single-bit variables, we can explicitly clear T0CS and PSA by writing:

```
T0CS = 0;                       // select timer mode
PSA = 0;                        // assign prescaler to Timer0
```

Setting PS<2:0> to 100 is more awkward. We could write:

```
PS2 = 1; PS1 = 0; PS0 = 0;      // PS=100 (prescale = 32)
```

But this generates three bit set/clear instructions (the code is inefficient), and it's not obvious that a value is being assigned to a bit field.

Another approach is to use the AND operator, with a bit mask, preserving the value of the upper five bits of the OPTION register, so that the value of PS<2:0> can be OR'ed in:

```
OPTION = OPTION & 0b11111000 | 0b100;   // PS=100 (prescale = 32)
```

Or equivalently:

```
OPTION &= 0b11111000 | 0b100;   // PS=100 (prescale = 32)
```

Which you use is largely a question of personal style – and you can adapt your style as appropriate. It is often preferable to use symbolic bit names to specify just one or two register bits, but using binary constants if several bits need to be specified at once, especially where some bits need to be set and others cleared (as is the case here), is quite acceptable – assuming that it is clearly commented, as above.

Using C macros

As explained in lesson 1, PICC-Lite comes with sample delay routines, including '`DelayMs()`', which provides a delay of up to 255 ms.

To create the initial delay of 2 s, using PICC-Lite, we could use eight successive '`DelayMs(250)`' calls.

To save space, we could use a loop, such as:

```
for (i = 0; i < 8; i++)
    DelayMs(250);
```

Or, since 20 × 100 ms = 2 seconds:

```
for (i = 0; i < 20; i++)
    DelayMs(100);
```

And then, at the end of the main loop, to create the final 1 s delay, we could use:

```
for (i = 0; i < 10; i++)
    DelayMs(100);
```

We are repeating essentially the same block of code, with a different end count in the 'for' loop.

As we saw in midrange lesson 5, the MPASM assembler provides a *macro* facility, which allows a parameterised segment of code to be defined once and then inserted multiple times into the source code. Macros are useful in this situation, where similar code blocks are repeated – especially for a block of code which implements a useful function such as a delay, since the macro can be reused in other programs.

C also allows macros to be defined.

For example, to implement a "delay in seconds" macro in PICC-Lite, we could use:

```
// Delay in seconds
//   Max delay is 25.5 sec
//   Calls: DelayMs() (defined in delay.h)
#define DelayS(T) {unsigned char i; for (i=0; i<T*10; i++) DelayMs(100);}
```

Having defined this macro, it can be used as if it was a function:

```
DelayS(2);                  // delay 2s
```

'`DelayS()`' could have been added, as either a function or a macro, to the existing "delay.h" and "delay.c" files. But if you modify those files, you would need to make it clear that these are your own customised versions, not the ones originally provided with PICC-Lite. It is better to create your own library of useful macros, which you would keep together in one or more header files, such as 'stdmacros.h', and reference using the `#include` directive.

Lesson 1 also explained that HI-TECH C PRO, when run in the free "Lite" mode, cannot use the sample delay code provided with PICC-Lite, because it does not optimise the code it generates – making the delays much longer than they should be. But we saw that that isn't a problem, because a built-in '`_delay()`' function and '`__delay_us()`' and '`__delay_ms()`' macros can be used to provide accurate delays.

This means that we would implement the "delay in seconds" macro for HI-TECH C PRO as:

```
// Delay in seconds
//   Max delay is 25.5 sec
//   Calls: DelayMs() (defined in delay.h)
#define DelayS(T) {unsigned char i; for (i=0; i<T*10; i++) __delay_ms(100);}
```

This '`DelayS()`' macro can then be used in exactly the same way as with PICC-Lite.

Thus, by encapsulating the compiler-specific code ('`DelayMs()`' versus '`__delay_ms()`') within a macro, the rest of the code, calling the macro, can be the same for the two compilers. This is a technique worth remembering, if you need to write portable code.

The TMR0 register is accessed through a variable, `TMR0`, so to clear it, we can write:

```
TMR0 = 0;                   // clear timer0
```

and to wait until 8 ms has elapsed:

```
while (TMR0 < 8000/32)      // wait for 8ms (32us/tick)
    ;
```

The "wait for button press or one second" routine can then the implemented as:

```
cnt8ms = 0;
while (BUTTON == 1 && cnt8ms < 1000/8) {
    TMR0 = 0;               // clear timer0
    while (TMR0 < 8000/32)  // wait for 8ms (32us/tick)
        ;
    ++cnt8ms;               // increment 8ms counter
}
```

(where, previously, 'BUTTON' had been defined as a symbol for 'GPIO3' – your code will be easier to maintain if you use symbolic names to refer to pins)

Finally, checking elapsed time is simply:

```
if (cnt8ms < MAXRT/8)       // if time < max reaction time (8ms/count)
    SUCCESS = 1;            //   turn on success LED
```

Complete program

Here is the complete reaction timer program, using HI-TECH C PRO, so that you can see how the various parts fit together:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 1                                 *
*                   Reaction Timer game.                                *
*                                                                       *
*   User must attempt to press button within defined reaction time      *
*   after "start" LED lights.  Success is indicated by "success" LED.   *
*                                                                       *
*       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                         *
*                                                                       *
************************************************************************/

#include <htc.h>

#define _XTAL_FREQ  4000000     // oscillator frequency for _delay()


/***** CONSTANTS *****/
#define MAXRT   200             // Maximum reaction time in ms


/***** CONFIGURATION *****/

// Pin assignments
#define START   GPIO2           // LEDs
#define SUCCESS GPIO1

#define BUTTON  GPIO3           // switches

// Config: int reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);
```

```
/***** MACROS *****/

// Delay in seconds
//   Max delay is 25.5 sec
//   Calls: DelayMs() (defined in delay.h)
#define DelayS(T) {unsigned char i; for (i=0; i<T*10; i++) __delay_ms(100);}


/***** MAIN PROGRAM *****/
void main()
{
    unsigned char   cnt8ms;         // 8ms counter (incremented every 8ms)

    // Initialisation
    TRISIO = 0b111001;              // configure GP1 and GP2 as outputs
    OPTION = 0b11010100;            // configure Timer0:
             //--0-----                 timer mode (T0CS = 0)
             //----0---                 prescaler assigned to Timer0 (PSA = 0)
             //-----100                 prescale = 32 (PS = 100)
             //                         -> increment every 32 us

    // Main loop
    for (;;) {
        GPIO = 0;                   // start with all LEDs off

        DelayS(2);                  // delay 2s

        START = 1;                  // turn on start LED

        // wait up to 1 sec for button press
        cnt8ms = 0;
        while (BUTTON == 1 && cnt8ms < 1000/8) {
            TMR0 = 0;               // clear timer0
            while (TMR0 < 8000/32)  // wait for 8ms (32us/tick)
                ;
            ++cnt8ms;               // increment 8ms counter
        }
        // check elapsed time
        if (cnt8ms < MAXRT/8)       // if time < max reaction time (8ms/count)
            SUCCESS = 1;            //   turn on success LED

        DelayS(1);                  // delay 1s

    }   // repeat forever
}
```

Comparisons

As we did in lesson 1, we can compare, for each language/compiler (MPASM assembler, HI-TECH PICC-Lite and C PRO), the length of the source code (ignoring comments and white space) versus program and data memory used by the resulting code. As a rough approximation, longer source code means more time spent by the programmer writing the code, and more time spent debugging or maintaining the code. The C source code tends to be much shorter than assembly code, while the C compilers tend to generate code that uses more memory than hand-crafted assembly does. Hence, these comparisons illustrate the trade-off between programmer efficiency and resource-usage efficiency.

The resource usage of the baseline (PIC12F509) versions of this example, from baseline C lesson 1, is also given for comparison:

Reaction_timer

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	56	53	54	56	4	4
HI-TECH PICC-Lite	25	25	76	89	7	11
HI-TECH C PRO Lite	24	24	111	109	7	4

As expected, the C source code is less than half as long as the assembler source, but the generated C program code is significantly larger (around 50% for the PICC-Lite compilers, which optimises the code it generates, but more than twice as large for HI-TECH C PRO, which does not perform any optimisation when running in "Lite" mode) and uses more data memory.

Also note that, because the midrange architecture is more suited to C programming, the PICC-Lite compiler is better able to optimise the code it generates for the 12F629, compared with the baseline 12F509.

Example 2: Background Process Timing

As discussed in midrange lesson 4, one of the key uses of timers is to provide regular timing for "background" processes, while a "foreground" process responds to user signals. Timers are ideal for this, because they continue to run, at a steady rate, regardless of any processing the PIC is doing. On midrange PICs this is normally done using timer-driven interrupts, which will be covered in the next lesson. However, the non-interrupt method, described in baseline C lesson 2, can still be used, and is covered here mainly for completeness.

The example in midrange lesson 4 used the circuit above, flashing the LED on GP2 at a steady 1 Hz, while lighting the LED on GP1 whenever the pushbutton is pressed.

The 500 ms delay needed for the 1 Hz flash was derived from Timer0 as follows:

- 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.

This was implemented by the following code:

```
;***** Main loop
loop    ; delay 500ms
        movlw   .125            ; repeat 125 times
        movwf   dlycnt          ; (125 x 4ms = 500ms)
dly500  ; (begin 500ms delay loop)
        banksel TMR0            ; clear timer0
        clrf    TMR0
w_tmr0      ; check timer0 until 4ms elapsed
        movf    TMR0,w
        xorlw   .125            ; (4ms = 125 x 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
        banksel GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop
```

And then the code which responds to the pushbutton was placed within the timer wait loop:

```
w_tmr0      ; check and respond to button press
        banksel GPIO
        bcf     sGPIO,GP1       ; assume button up -> LED off
        btfss   GPIO,GP3        ; if button pressed (GP3 low)
        bsf     sGPIO,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 x 32us)
        btfss   STATUS,Z
        goto    w_tmr0
```

The additional code doesn't affect the timing of the background task (flashing the LED), because there are only a few additional instructions; they are able to be executed within the 32 µs available between each "tick" of Timer0.

HI-TECH C Implementation

There are no new features to introduce; Timer0 is setup and accessed in the same way as in the last example.

Here is one way that the program logic, equivalent to the assembly code above, can be implemented in C:

```
    // Main loop
    for (;;) {
        // delay 500ms while checking for button press
        for (dc = 0; dc < 125; dc++) {  // repeat for 500ms (125 x 4ms = 500ms)
            TMR0 = 0;                       // clear timer0
            while (TMR0 < 125) {            // repeat for 4ms (125 x 32us)
                sGPIO &= ~(1<<1);               // assume button up -> LED off
                if (GPIO3 == 0)                 // if button pressed (GP3 low)
                    sGPIO |= 1<<1;              //   turn on LED on GP1

                GPIO = sGPIO;                   // update GPIO
            }
        }
        // toggle LED on GP2
        sGPIO ^= 1<<2;

    }   // repeat forever
```

Note the syntax used to set, clear and toggle bits in the shadow GPIO variable, `sGPIO`:

```
sGPIO |= 1<<1;              // turn on LED on GP1
sGPIO &= ~(1<<1);           // turn off LED on GP1
sGPIO ^= 1<<2;              // toggle LED on GP2
```

We could instead have written:

```
sGPIO |= 0b00000010;        // turn on LED on GP1
sGPIO &= 0b11111101;        // turn off LED on GP1
sGPIO ^= 0b00000100;        // toggle LED on GP2
```

However, the right shift ('<<') form more clearly specifies which bit is being operated on.

Note also that there no need to update GPIO after the LED on GP2 is toggled, because GPIO is being continually updated from `sGPIO` within the inner timer wait loop.

Comparisons

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

Flash+PB_LED

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	42	37	36	31	2	2
HI-TECH PICC-Lite	18	18	38	46	4	6
HI-TECH C PRO Lite	18	18	67	65	5	3

The PICC-Lite compiler does particularly well in this example, once again generating more efficient code for the 12F629 than for the 12F509 – in this case, almost as small as the hand-written assembler version, while the C source code remains less than half as long as the assembler source.

Example 3: Switch debouncing

The previous lesson demonstrated one method commonly used to debounce switches: sampling the switch state periodically, and only considering it to have definitely changed when it has been in the new state for some minimum number of successive samples.

This "counting algorithm" was given as:

```
count = 0
while count < max_samples
      delay sample_time
      if input = required_state
            count = count + 1
      else
            count = 0
end
```

As explained in midrange lesson 4, this can be simplified by using a timer, since the timer increments automatically:

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

This algorithm was implemented in assembler, to wait for and debounce a "button down" event, as follows:

```
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
```

This code assumes that Timer0 is available, and is in timer mode, with a 1 MHz instruction clock and a 1:64 prescaler, giving 64 µs per tick.

Of course, since the baseline PICs only have a single timer, it is likely that Timer0 is being used for something else, and so is not available for switch debouncing. But if it is available, it makes sense to use it.

This was demonstrated by applying this timer-based debouncing method to the "toggle an LED on pushbutton press" program developed in midrange lesson 3.

HI-TECH C Implementation

Timer0 can be configured for timer mode, with a 1:64 prescaler, by:

```
    OPTION = 0b11010101;            // configure Timer0:
             //--0-----                 timer mode (T0CS = 0)
             //----0---                 prescaler assigned to Timer0 (PSA = 0)
             //-----101                 prescale = 64 (PS = 101)
             //                         -> increment every 64 us
```

This is the same as for the 1:32 prescaler examples, above, except that the PS<2:0> bits are set to '101' instead of '100'.

The timer-based debounce algorithm, given above in pseudo-code, is readily translated into C:

```
        TMR0 = 0;                   // reset timer
        while (TMR0 < 157)          // wait at least 10ms (157 x 64us = 10ms)
            if (GPIO3 == 1)         //   if button up,
                TMR0 = 0;           //     restart wait
```

This could be defined as a macro (to be placed in a header file) as follows:

```
#define DEBOUNCE 10*1000/256    // switch debounce count = 10ms/(256us/tick)

// DbnceLo()
//
// Debounce switch on given input pin
// Waits for switch input to be high continuously for 10ms
//
// Uses: TMR0       Assumes: TMR0 running at 256us/tick
//
#define DbnceLo(PIN) TMR0 = 0;                  /* reset timer              */ \
                     while (TMR0 < DEBOUNCE)    /* wait until debounce time */ \
                         if (PIN == 1)          /*   if input high,         */ \
                             TMR0 = 0           /*     restart wait         */
```

and then called from the main program as, for example:

```
        DbnceLo(GPIO3);  // wait until button pressed (GP3 low)
```

Complete program

Here is how this timer-based debounce code (without using macros) fits into the HI-TECH C version of the "toggle an LED on pushbutton press" program:

```
/************************************************************************
*   Description:    Lesson 2, example 3a                                *
*                                                                       *
*   Demonstrates use of Timer0 to implement debounce counting algorithm *
*                                                                       *
*   Toggles LED when pushbutton is pressed (low) then released (high)   *
************************************************************************
*   Pin assignments:                                                    *
*       GP1 - flashing LED                                              *
*       GP3 - pushbutton switch                                         *
************************************************************************/

#include <htc.h>

// Config: int reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);

void main()
{
    unsigned char   sGPIO;            // shadow copy of GPIO

    // Initialisation
    GPIO = 0;                         // start with LED off
    sGPIO = 0;                        //   update shadow
    TRISIO = 0b111101;                // configure GP1 (only) as an output
    OPTION = 0b11010101;              // configure Timer0:
             //--0-----                    timer mode (T0CS = 0)
             //----0---                    prescaler assigned to Timer0 (PSA = 0)
             //-----101                    prescale = 64 (PS = 101)
             //                            -> increment every 64 us

    // Main loop
    for (;;) {
        // wait until button pressed (GP3 low), debounce using timer0:
        TMR0 = 0;                     // reset timer
        while (TMR0 < 157)            // wait at least 10ms (157 x 64us = 10ms)
            if (GPIO3 == 1)           //   if button up,
                TMR0 = 0;             //     restart wait

        // toggle LED on GP1
        sGPIO ^= 1<<1;                // flip shadow GP1
        GPIO = sGPIO;                 // write to GPIO

        // wait until button released (GP3 high), debounce using timer0:
        TMR0 = 0;                     // reset timer
        while (TMR0 < 157)            // wait at least 10ms (157 x 64us = 10ms)
            if (GPIO3 == 0)           //   if button down,
                TMR0 = 0;             //     restart wait

    }   // repeat forever
}
```

Comparisons

Here is the resource usage summary for the "toggle an LED using timer-based debounce" programs:

Timer_debounce

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	42	35	37	30	1	1
HI-TECH PICC-Lite	19	19	36	45	3	5
HI-TECH C PRO Lite	19	19	72	70	4	3

Once again, the C source code is less than half as long as the assembler version, while the PICC-Lite compiler generates particularly efficient code – even smaller than the hand-written assembler equivalent! This is possible because some of the bank selection directives in the assembler version were not strictly needed. The C compiler can keep track of which bank is selected, avoiding unnecessary instructions.

The code generated by the C compilers is also significantly more compact, and using less data memory, than that generated for the corresponding example in lesson 1, where delay functions were used in implementing a counter-based debounce algorithm. For example, the PICC-Lite version of the delay+counter debounce program required 64 words of program memory, compared with 36 words for the timer-based program.

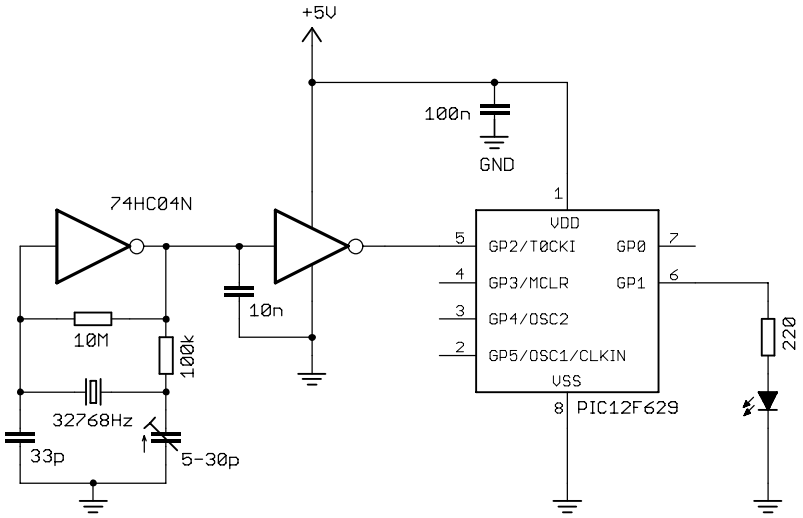
The lesson here is that your code can be shorter and more efficient if you are able to use a timer (or, as we'll see in the next lesson, a timer-based interrupt) for switch debouncing.

Example 4: Using Counter Mode

The previous three examples use Timer0 in "timer mode", where it is clocked by the PIC's instruction clock, which runs at ¼ the speed of the processor clock (i.e. a nominal 1 MHz when the nominally 4 MHz internal RC oscillator is used).

As we saw in midrange lesson 4, the timer can instead be used in "counter mode", where it counts transitions (rising or falling) on the PIC's T0CKI input.

We can use the example from that lesson to illustrate how Timer0 can be used as a counter, using C: Timer0 is driven by an external 32.768 kHz crystal oscillator (as shown on the right), providing a time base that can be used to flash an LED at a reasonably accurate 1 Hz.



If the 32.768 kHz clock input is divided (prescaled) by 128, bit 7 of TMR0 will cycle at 1 Hz.

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

This was done in midrange lesson 4 by:

```
        movlw   b'11110110'     ; configure Timer0:
                ; --1-----          counter mode (T0CS = 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
```

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

Bit 7 of TMR0 (which is cycling at 1 Hz) was then continually copied to GP1 (using a shadow register), as follows:

```
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
```

HI-TECH C Implementation

As always, to configure Timer0 using HI-TECH C, simply assign the appropriate value to `OPTION`:

```
    OPTION = 0b11110110;        // configure Timer0:
             //--1-----             counter mode (T0CS = 1)
             //----0---             prescaler assigned to Timer0 (PSA = 0)
             //-----110             prescale = 128 (PS = 110)
             //                     -> increment at 256 Hz with 32.768 kHz input
```

To test bit 7 of TMR0, we can use the following construct:

```
        if (TMR0 & 1<<7)            // if TMR0<7>=1
            sGPIO |= 1<<1;          //   turn on LED
```

This works because the expression "1<<7" equals 10000000 binary; the result of ANDing TMR0 with 1<<7 will be non-zero only if TMR0<7> is set.

Complete program

Here is the complete "flash an LED using crystal-driven timer" program:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 4                                 *
*                                                                       *
*   Demonstrates use of Timer0 in counter mode                          *
*                                                                       *
*   LED flashes at 1 Hz (50% duty cycle),                               *
*   with timing derived from 32.768 kHz input on T0CKI                  *
*                                                                       *
************************************************************************
*                                                                       *
```

```
*   Pin assignments:                                                    *
*       GP1 - flashing LED                                              *
*       T0CKI - 32.768 kHz signal                                       *
*                                                                       *
************************************************************************/

#include <htc.h>

// Config: ext reset, no code protect, no brownout detect, no watchdog,
//         power-up timer enabled, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO);


void main()
{
    unsigned char   sGPIO;      // shadow copy of GPIO

    // Initialisation
    TRISIO = ~(1<<1);           // configure GP1 (only) as an output
    OPTION = 0b11110110;        // configure Timer0:
             //--1-----             counter mode (T0CS = 1)
             //----0---             prescaler assigned to Timer0 (PSA = 0)
             //-----110             prescale = 128 (PS = 110)
             //                     -> increment at 256 Hz with 32.768 kHz input

     // Main loop
    for (;;)
    {
        // TMR0<7> cycles at 1Hz so continually copy to GP1
        sGPIO = 0;                  // assume TMR<7>=0 -> LED off (shadow)
        if (TMR0 & 1<<7)            // if TMR0<7>=1
            sGPIO |= 1<<1;          //   turn on LED (shadow)

        GPIO = sGPIO;               // copy shadow to GPIO

    }   // repeat forever
}
```

Comparisons

Here is the resource usage summary for the "flash an LED using a crystal-driven timer" programs:

Timer_debounce

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	12F509	12F629	12F509	12F629	12F509
Microchip MPASM	25	20	20	15	1	1
HI-TECH PICC-Lite	11	11	19	29	3	5
HI-TECH C PRO Lite	11	11	26	24	3	2

Again, the PICC-Lite compiler is able to generate very efficient code for 12F629 in this example – significantly smaller than the corresponding code generated by the same compiler for the 12F509, and even smaller than the hand-written assembler version!

Summary

These examples demonstrate that Timer0 can be effectively configured and accessed using the HI-TECH C compilers. The program algorithms can often be expressed quite succinctly in C, as illustrated by the code length comparisons:

Source code (lines)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4
Microchip MPASM	56	42	42	25
HI-TECH PICC-Lite	25	18	19	11
HI-TECH C PRO Lite	24	18	19	11

The C source code is consistently less than half the length of the corresponding assembler source.

Although the C compilers generated significantly larger code than assembler in example 1, the PICC-Lite compiler was able to generate highly optimised code for the other examples – in some cases bettering the assembler versions:

Program memory (words)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4
Microchip MPASM	54	36	37	20
HI-TECH PICC-Lite	76	38	36	19
HI-TECH C PRO Lite	111	67	72	26

As mentioned above, the hand-written assembler code could be optimised further by removing redundant `banksel` directives. This would, however, make the code less maintainable. The C compilers manage bank selection for you – one of the nice aspects about using C for midrange PICs.

The programs generated by the C compilers do, however, consistently use more data memory than the assembler versions:

Data memory (bytes)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4
Microchip MPASM	4	2	1	1
HI-TECH PICC-Lite	7	4	3	3
HI-TECH C PRO Lite	7	5	4	3

Of course this is not an important issue in these small examples, where, even in example 1, the C programs are using only 7 out of 64 bytes of data memory available on the 12F629.

In the next lesson we'll see how interrupts can be implemented using HI-TECH C.