Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 2: Using Timer 0

As demonstrated in the previous lesson, C can be a viable choice for programming digital I/O operations on baseline (12-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 baseline lesson 5 (which you should refer to while working through this tutorial) 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 bundled with Microchip's MPLAB: HI-TECH C[1] (in "Lite" mode), PICC-Lite and CCS PCB, 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 baseline lesson 6, including macros.

In summary, this lesson covers:

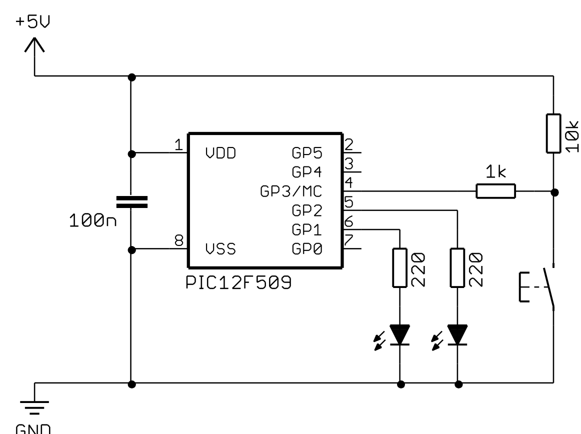
- Configuring Timer0 as a timer or counter
- Accessing Timer0
- Using Timer0 for switch debouncing
- Using C macros

with examples for HI-TECH C, PICC-Lite and CCS PCB.

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, baseline lesson 5 included an example "reaction timer" game, based on 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 (that is, within some 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

[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.

mode (clocked by the PIC's instruction clock, which in this example is 1 MHz).

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 represented 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 baseline lesson 5, Timer0 is configured so that it is clocked every 32 µs, 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 is effectively measuring time in 8 ms intervals. This "8 ms counter" can then be checked, when the pushbutton is pressed, to see whether the maximum reaction time has been exceeded.

As explained in that lesson, 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'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
```

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

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

(This code is actually taken from baseline lesson 6)

HI-TECH PICC-Lite

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

```
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
```

Note that this has been commented in a way which documents which bits affect each setting, with '`-`'s indicating "don't care".

However, some purists would argue, for both assembler and C, that we should be using symbols (defined in the include, or *header* files), instead of binary constants.

Since the intent is to clear T0CS and PSA, and to set PS<2:0> to 100, we could make that intent explicit by writing:

```
OPTION = ~T0CS & ~PSA & 0b11111000 | 0b100;
```

('`0b11111000`' is used to mask off the lower three bits, so that the value of PS<2:0> can be OR'ed in.)

Alternatively, we could write:

```
OPTION = ~T0CS & ~PSA | PS2 & ~PS1 & ~PS0;
```

(specifying the individual PS<2:0> bits)

Which you use is largely a question of personal style – and you can adapt your style as appropriate. Although it is often preferable to use symbolic bit names to specify just one or two register bits, using binary constants is quite acceptable 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) – 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, we could use eight successive '`DelayMs(250)`' calls (8 × 250 ms = 2 s). Or, 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 baseline lesson 6, 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 type of 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.

Macros can also be used when programming in C.

For example, to implement a "delay in seconds" macro, 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, as was done in baseline lesson 6, 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.

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 '`GP3`' – as discussed in lesson 6, 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 PICC-Lite, 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   4MHZ        // oscillator frequency for DelayMs()
#include "delay.h"              // defines DelayMs()


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


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

// Pin assignments
#define START   GP2             // LEDs
#define SUCCESS GP1

#define BUTTON  GP3             // switches

// Config: int reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & WDTDIS & INTRC);


/***** 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++) DelayMs(100);}


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

    // Initialisation
    TRIS = 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
}
```

HI-TECH C Pro Lite

As we saw in lesson 1, the HI-TECH C PRO compiler provides '`__delay_us()`' and '`__delay_ms()`' macros, which, to generate accurate delays, should be used instead of the example delay code that had been included with PICC-Lite.

To modify the PICC-Lite program, above, to use these HI-TECH C Pro macros, we need to first change the processor frequency definition from:

```
#define XTAL_FREQ   4MHZ        // oscillator frequency for DelayMs()
```

to:

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

We can then change the definition of the '`DelayS()`' macro from:

```
#define DelayS(T) {unsigned char i; for (i=0; i<T*10; i++) DelayMs(100);}
```

to:

```
#define DelayS(T) {unsigned char i; for (i=0; i<T*10; i++) __delay_ms(100);}
```

CCS PCB

Lesson 1 introduced the '`setup_counters()`' function, which, as we saw, has to be used if you need to enable or disable the weak pull-ups. But its primary purpose (hence the name "setup counters") is to setup Timer0 and the watchdog timer.

To configure Timer0 for timer mode (using the internal instruction clock), with the prescaler set to 1:32 and assigned to Timer0, you could use:

```
        setup_counters(RTCC_INTERNAL,RTCC_DIV_32);
```

However, CCS is de-emphasising the use of '`setup_counters()`', in favour of more specific timer and watchdog setup functions, including '`setup_timer_0()`':

```
        setup_timer_0(RTCC_INTERNAL|RTCC_DIV_32);
```

Note that '`setup_counters()`' takes two parameters, while '`setup_timer_0()`' takes a single parameter formed by OR'ing the two symbols.

Both functions work correctly, but '`setup_timer_0()`' produces smaller code, since it is only configuring Timer0, so it is the better choice here.

As we saw in the previous lesson, the CCS approach is to expose the PIC's functionality through built-in functions, instead of accessing the registers directly.

To set Timer0 to a specific value, use the '`set_timer0()`' function, for example:

```
        set_timer0(0);                  // clear timer0
```

To read the current value of Timer0, use the ‘get_timer0()’ function, for example:

```
        while (get_timer0() < 8000/32)  // wait for 8ms (32us/tick)
```

The code is then otherwise the same as for HI-TECH C.

There is no need to use a delay macro to create a 2 s delay, as we did with HI-TECH C, because the built-in CCS function, ‘delay_ms()’, can create a delay up to 65535 ms (65.5 s). However, if you wanted to create a “delay in seconds” macro, for consistency, you could simply use:

```
#define DelayS(T) delay_ms(1000*T)
```

Complete program

Here is the complete reaction timer program, using CCS PCB:

```
/************************************************************************
*                                                                       *
*   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 <12F509.h>
#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5
#use delay (clock=4000000)      // oscillator frequency for delay_ms()

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

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

// Pin assignments
#define START   GP2             // LEDs
#define SUCCESS GP1

#define BUTTON  GP3             // switches

// Config: int reset, no code protect, no watchdog, 4MHz int clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC


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

```
    // Initialisation
    // configure Timer0: timer mode, prescale = 32 (increment every 32us)
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_32);


    // Main loop
    while (TRUE) {
        output_b(0);                // start with all LEDs off

        delay_ms(2000);             // delay 2s

        output_high(START);         // turn on start LED

        // wait up to 1 sec for button press
        cnt8ms = 0;
        while (input(BUTTON) == 1 && cnt8ms < 1000/8) {
            set_timer0(0);                  // clear timer0
            while (get_timer0() < 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)
            output_high(SUCCESS);   //   turn on success LED

        delay_ms(1000);             // 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 and CCS PCB), 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.

Here is the resource usage summary for the "Reaction timer" programs:

Reaction_timer

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	52	55	4
HI-TECH PICC-Lite	25	89	11
HI-TECH C PRO Lite	24	109	4
CCS PCB	22	84	6

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 and CCS compilers, which optimise the code they generate, but nearly twice as large for HI-TECH C PRO, which does not perform any optimisation when running in "Lite" mode) and the PICC-Lite version in particular uses much more data memory.

Example 2: Background Process Timing

As discussed in baseline lesson 5, 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 more advanced PICs, a timer is generally used with an interrupt routine, to run these background tasks. But as we'll see, they can still be useful for maintaining the timing of background tasks, even without interrupts.

The example in baseline lesson 5 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:

```
start   ; delay 500ms
        banksel dlycnt
        movlw   .125            ; repeat 125 times
        movwf   dlycnt          ; -> 125 x 4ms = 500ms
dly500  clrf    TMR0            ; clear timer0
w_tmr0  movf    TMR0,w
        xorlw   .125            ; wait for 4ms (125x32us)
        btfss   STATUS,Z
        goto    w_tmr0
        decfsz  dlycnt,f        ; end 500ms delay loop
        goto    dly500

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

        ; repeat forever
        goto    start
```

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

```
w_tmr0      ; check and respond to button press
        bcf     sGPIO,1         ; 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
            ; check timer0 until 4ms elapsed
        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 PRO or PICC-Lite

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:

```
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 (GP3 == 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
```

But 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.

CCS PCB

Again, there are no new features to introduce. All we need do is to convert the references to GPIO and TMR0 in the HI-TECH PICC code into the CCS PCB built-in function equivalents:

```
while (TRUE) {
    // delay 500ms while checking for button press
    for (dc = 0; dc < 125; dc++) {  // repeat for 500ms (125 x 4ms = 500ms)
        set_timer0(0);                  // clear timer0
        while (get_timer0() < 125) {    // repeat for 4ms (125 x 32us)
            sGPIO &= ~(1<<1);               // assume button up -> LED off
            if (input(GP3) == 0)            // if button pressed (GP3 low)
                sGPIO |= 1<<1;              //   turn on LED on GP1

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

}   // repeat forever
```

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)
Microchip MPASM	35	30	2
HI-TECH PICC-Lite	18	46	6
HI-TECH C PRO Lite	18	65	3
CCS PCB	17	47	6

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 expressed as:

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

As explained in baseline lesson 5, this can be simplified by using a timer, since it 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,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
```

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 baseline lesson 4.

HI-TECH C PRO or PICC-Lite

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 (GP3 == 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         */
```

Note that a backslash ('\') is placed at the end of all but the last line, to continue the macro definition over multiple lines. To make the backslashes visible to the C pre-processor, the older "`/* */`" style comments must be used, instead of the newer "`//`" style.

This macro can then be called from the main program as, for example:

```
        DbnceLo(GP3); // 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 watchdog, 4MHz int clock
__CONFIG(MCLRDIS & UNPROTECT & WDTDIS & INTRC);


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

    // Initialisation
    GPIO = 0;                       // start with LED off
    sGPIO = 0;                      //   update shadow
    TRIS = ~(1<<1);                 // 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 (GP3 == 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 (GP3 == 0)           //   if button down,
                TMR0 = 0;           //     restart wait

    }   // repeat forever
}
```

CCS PCB

To configure Timer0 for timer mode with a 1:64 prescaler, using CCS PCB, use:

```
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_64);
```

This is the same as we have seen before, except with ‘`RTCC_DIV_64`’ instead of ‘`RTCC_DIV_32`’.

The timer-based debounce algorithm can then be expressed as:

```
        set_timer0(0);                  // reset timer
        while (get_timer0() < 157)      // wait at least 10ms (157 x 64us = 10ms)
            if (input(GP3) == 1)        //   if button up,
                set_timer0(0);          //     restart wait
```

In the same way as for HI-TECH C, this could instead be defined as a macro, as follows:

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

#define DbnceLo(PIN)                                                        \
    set_timer0(0);                        /* reset timer               */ \
    while (get_timer0() < DEBOUNCE)       /* wait until debounce time */ \
        if (input(PIN) == 1)              /* if input high,            */ \
            set_timer0(0)                 /*   restart wait            */
```

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

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

Complete program

Here is how the timer-based debounce code (without using macros) fits into the CCS PCB 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 <12F509.h>
#define GP0 PIN_B0              // define GP pins
#define GP1 PIN_B1
#define GP2 PIN_B2
#define GP3 PIN_B3
#define GP4 PIN_B4
#define GP5 PIN_B5

// Config: int reset, no code protect, no watchdog, 4MHz int clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC


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

    // Initialisation
    output_b(0);                    // start with LED off
    sGPIO = 0;                      //   update shadow
    // configure Timer0: timer mode, prescale = 64 (increment every 64us)
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_64);

    // Main loop
    while (TRUE)
    {
```

```
        // wait until button pressed (GP3 low), debounce using timer0:
        set_timer0(0);                  // reset timer
        while (get_timer0() < 157)      // wait at least 10ms (157 x 64us = 10ms)
            if (input(GP3) == 1)        //   if button up,
                set_timer0(0);          //     restart wait

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

        // wait until button released (GP3 high), debounce using timer0:
        set_timer0(0);                  // reset timer
        while (get_timer0() < 157)      // wait at least 10ms (157 x 64us = 10ms)
            if (input(GP3) == 0)        //   if button down,
                set_timer0(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)
Microchip MPASM	33	29	1
HI-TECH PICC-Lite	19	45	5
HI-TECH C PRO Lite	19	70	3
CCS PCB	18	52	5

This follows the pattern we've come to expect, where the C source code is only around half as long as the assembler version, while the optimising C compilers generate somewhat (around 50%) larger code than the hand-written assembler equivalent.

But in all cases, the code is more compact (for the C compilers, significantly more compact) than for the corresponding example in lesson 1, where delay functions were used in implementing a counter-based debounce algorithm – the extreme case is PICC-Lite, where the delay/counter debounced program used nearly twice the resources (77 words of program memory and 11 data registers) of the timer-based program.

The lesson here is that your code will be shorter and more efficient if you can use a timer for switch debouncing.

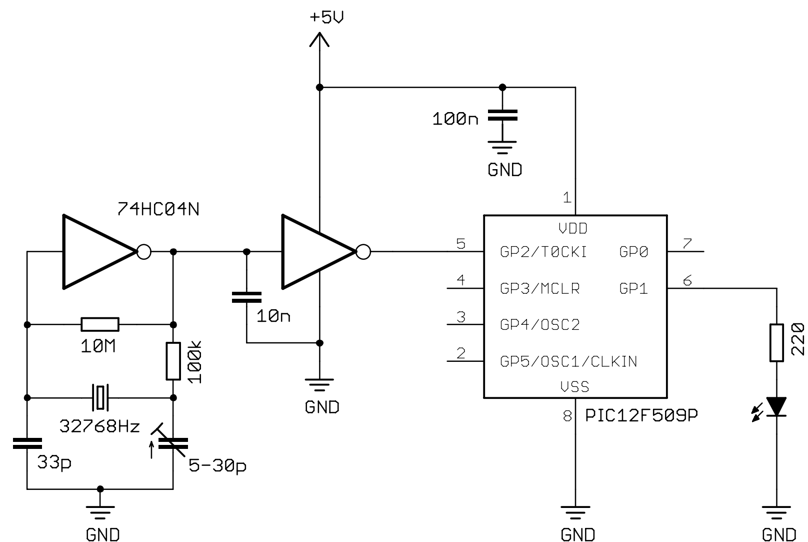
Example 4: Using Counter Mode

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 discussed in baseline lesson 5, the timer can instead be used in "counter mode", where it counts transitions (rising or falling) on the PIC's T0CKI input.

To illustrate how to use Timer0 as a counter, using C, we can use the example from baseline lesson 5: using an external 32.768 kHz crystal oscillator (as shown on the right) to drive the counter, providing a time base that can be used to flash an LED at a more 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 lesson 5 by:

```
        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
```

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, as follows:

```
start   ; 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    start
```

HI-TECH C PRO or PICC-Lite

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, so the result of ANDing TMR0 with 1<<7 will only be non-zero if TMR0<7> is set.

Complete program

Here is the HI-TECH C version of the "flash an LED using crystal-driven timer" program:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 4                                 *
*                                                                       *
*   Demonstrates use of Timer0 in counter mode                          *
*                                                                       *
*   LED flashes at 1Hz (50% duty cycle),                                *
*   with timing derived from 32.768KHz input on T0CKI                   *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 - flashing LED                                              *
*       T0CKI - 32.768kHz signal                                        *
*                                                                       *
************************************************************************/

#include <htc.h>

// Config: ext reset, no code protect, no watchdog, 4MHz int clock
__CONFIG(MCLREN & UNPROTECT & WDTDIS & INTRC);


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

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

        GPIO = sGPIO;               // copy shadow to GPIO

    }   // repeat forever
}
```

CCS PCB

To configure Timer0 for counter mode, instead of timer mode, using the CCS PCB '`setup_timer_0()`' function, use either '`RTCC_EXT_L_TO_H`' (to count low to high input transitions on T0CKI), or '`RTCC_EXT_H_TO_L`', (for high to low transitions), instead of '`RTCC_INTERNAL`' (which specifies timer mode).

For example, in this case:

```
    setup_timer_0(RTCC_EXT_L_TO_H|RTCC_DIV_128);
```

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

```
        if (get_timer0() & 1<<7)     // if TMR0<7>=1
            sGPIO |= 1<<1;           //   turn on LED
```

However, CCS PCB does provide a facility for accessing (testing or setting/clearing) bits directly.

The bit to be accessed must first be declared as a variable, using the `#bit` directive.

For example:

```
#bit TMR0_7 = 0x01.7                 // bit 7 of TMR0
```

This variable can then be used the same way as any other single-bit variable, and can be tested directly.

For example:

```
        if (TMR0_7)                  // if TMR0<7>=1
            sGPIO |= 1<<1;           //   turn on LED
```

Or, for clarity, you may prefer to write:

```
        if (TMR0_7 == 1)             // if TMR0<7>=1
```

But for testing a status bit, most programmers will readily understand the '`if (TMR0_7)`' form.

As you can see, defining bit variables in this way makes for straightforward, easy-to-read code. However, CCS discourage this practice; as the help file warns, "*Register locations change between chips*". For example, the code above assumes that TMR0 is located at address 0x01. If this code is migrated to a PIC with a different address for TMR0, and you forget to change the bit variable definition, the problem may be very hard to find – the compiler wouldn't produce an error. Your code would simply continue to test bit 7 of whatever register happened to now be at address 0x01.

So although, by using the `#bit` directive, you can make your code clearer and more efficient, you should use it carefully. Using the CCS built-in functions is safer, and easier to maintain.

Complete program

Here is how the code, using CCS PCB, with the `#bit` pre-processor directive, for the "flash an LED using crystal-driven timer" fits together:

```
/************************************************************************
*                                                                       *
*   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 <12F509.h>

#bit TMR0_7 = 0x01.7                // bit 7 of TMR0

// Config: ext reset, no code protect, no watchdog, 4 MHz int clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC


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

    // Initialisation
    // configure Timer0: counter mode, prescale = 128
    //   (increment at 256 Hz with 32.768 kHz input)
    setup_timer_0(RTCC_EXT_L_TO_H|RTCC_DIV_128);

    // Main loop
    while (TRUE)
    {
        // TMR0<7> cycles at 1 Hz
        // so continually copy to GP1
        sGPIO = 0;                  // assume TMR<7>=0 -> LED off
        if (TMR0_7)                 // if TMR0<7>=1
            sGPIO |= 1<<1;          //   turn on LED

        output_b(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:

Flash_LED_XTAL

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	18	14	1
HI-TECH PICC-Lite	11	29	5
HI-TECH C PRO Lite	11	24	2
CCS PCB	11	18	4

The code generated by the CCS compiler for this example is very efficient, needing only slightly more program memory than the hand-written assembler equivalent.

And, unusually, the code generated by HI-TECH C PRO's "Lite" mode is actually smaller than the "optimised" code generated by PICC-Lite.

Summary

These examples have demonstrated that Timer0 can be effectively configured and accessed using the HI-TECH and CCS C compilers, with the program algorithms often being able to 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	52	35	33	18
HI-TECH PICC-Lite	25	18	19	11
HI-TECH C PRO Lite	24	18	19	11
CCS PCB	22	17	18	11

Once again, the C source code is generally around half the length of the corresponding assembler source, and the CCS source code is consistently a little shorter than that for HI-TECH C, reflecting the availability of built-in functions in CCS PCB.

We have also seen that all of the C compilers generate code which generally occupies significantly more program memory and uses more data memory than for corresponding hand-written assembler programs:

Program memory (words)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4
Microchip MPASM	55	30	29	14
HI-TECH PICC-Lite	89	46	45	29
HI-TECH C PRO Lite	109	65	70	24
CCS PCB	84	47	52	18

Data memory (bytes)

Assembler / Compiler	Example 1	Example 2	Example 3	Example 4
Microchip MPASM	4	2	1	1
HI-TECH PICC-Lite	11	6	5	5
HI-TECH C PRO Lite	4	3	3	2
CCS PCB	6	6	5	4

The exception is the CCS compiler in example 4, where the compiler-generated code is only a little larger than assembler, demonstrating that the C compilers can, in some circumstances, be very efficient – even on the baseline PICs.

In the next lesson we'll continue our review of past topics, seeing how these C compilers can be used with sleep mode and the watchdog timer, and to select various clock, or oscillator, configurations.