Introduction to PIC Programming

Programming Baseline PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 3: Using Timer 0

As demonstrated in the previous lessons, 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[1]: HI-TECH C (in "Lite" mode) and CCS PCB, introduced in lesson 1. 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 and CCS PCB.

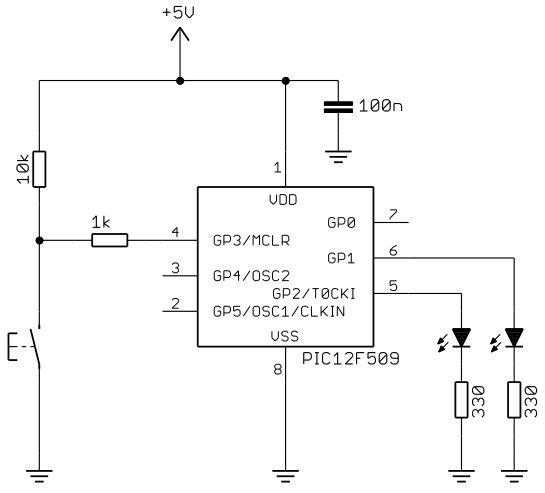
Note that this tutorial series assumes a working knowledge of the C language; it does **not** attempt to teach C.

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.

To implement this circuit using the Gooligum baseline training board, connect jumpers JP3, JP12 and JP13 to enable the pull-up resistor on GP3 and the LEDs on GP1 and GP2.

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



[1] At the time of writing (Feb 2012), MPLAB 8 is bundled with both CCS PCB and HI-TECH C, while MPLAB X is bundled with HI-TECH C only. You should download the latest version of HI-TECH C from www.microchip.com.

The pushbutton has to be pressed as quickly as possible after the LED on GP2 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.

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).

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

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

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 used to implement the button press / timing test routine:

```
        ; wait up to 1 sec for button press
        banksel cnt_8ms             ; clear timer (8 ms counter)
        clrf    cnt_8ms             ; repeat for 1 sec:
wait1s  clrf    TMR0                ;   clear Timer0
w_tmr0                              ;   repeat for 8 ms:
        btfss   BUTTON              ;     if button pressed (low)
        goto    btn_dn              ;       finish delay loop immediately
        movf    TMR0,w
        xorlw   8000/32             ;   (8 ms at 32 us/tick)
        btfss   STATUS,Z
        goto    w_tmr0
        incf    cnt_8ms,f           ;   increment 8 ms counter
        movlw   1000/8              ; (1 sec at 8 ms/count)
        xorwf   cnt_8ms,w
        btfss   STATUS,Z
        goto    wait1s
        ; check elapsed time
btn_dn  movlw   MAXRT/8             ; if time < max reaction time (8 ms/count)
        subwf   cnt_8ms,w
        btfss   STATUS,C
        bsf     SUCCESS             ;   turn on success LED
```

(This code is actually taken from baseline lesson 6)

HI-TECH C

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".

Alternatively, you could express that using the symbols defined in the processor header file ('pic12f509.h' in this case).

For example, 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.)

Or, you could write:

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

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

Which approach 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.

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 8 ms (32 us/tick)
    ;
```

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

```
cnt_8ms = 0;
while (BUTTON == 1 && cnt_8ms < 1000/8)
{
    TMR0 = 0;               // clear timer0
    while (TMR0 < 8000/32)  // wait for 8 ms (32 us/tick)
        ;
    ++cnt_8ms;              // increment 8 ms counter
}
```

where, previously, '`BUTTON`' had been defined as a symbol for '`GP3`'.

As discussed in baseline lesson 6, your code will be easier to understand and maintain if you use symbolic names to refer to pins. If your design changes, you can update the definitions in one place (usually placed at the start of your c, or in a header file). Of course, you may also need to modify your initialisation statements, such as '`TRIS =`'. This is a good reason to keep all your initialisation code in one easily-found place, such as at the start of the program, or in an "init()" function.

Finally, checking elapsed time is simply:

```
        if (cnt_8ms < MAXRT/8)      // if time < max reaction time (8 ms/count)
            SUCCESS = 1;            //   turn on success LED
```

Complete program

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

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 3, 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                         *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = success LED                                               *
*       GP2 = start LED                                                 *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <htc.h>
#include <stdint.h>


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);

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

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

#define BUTTON  GP3             // pushbutton


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


/***** MAIN PROGRAM *****/
void main()
{
    uint8_t     cnt_8ms;            // counter: increments every 8 ms

    // Initialisation
    TRIS = 0b111001;                // configure GP1 and GP2 (only) 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 (;;)
    {
        // start with both LEDs off
        GPIO = 0;

        // delay 2 sec
        __delay_ms(2000);           // delay 2000 ms

        // indicate start
        START = 1;                  // turn on start LED

        // wait up to 1 sec for button press
        cnt_8ms = 0;
        while (BUTTON == 1 && cnt_8ms < 1000/8)
        {
            TMR0 = 0;               // clear timer0
            while (TMR0 < 8000/32)  // wait for 8 ms (32 us/tick)
                ;
            ++cnt_8ms;              // increment 8 ms counter
        }
        // check elapsed time
        if (cnt_8ms < MAXRT/8)      // if time < max reaction time (8 ms/count)
            SUCCESS = 1;            //   turn on success LED

        // delay 1 sec
        __delay_ms(1000);           // delay 1000 ms

    }   // repeat forever
}
```

CCS PCB

Lesson 2 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 have seen in the previous lessons, 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.

Complete program

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

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 3, 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                         *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = success LED                                               *
*       GP2 = start LED                                                 *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#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


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

#use delay (clock=4000000)      // oscillator frequency for delay_ms()

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

#define BUTTON  GP3             // pushbutton
```

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


/***** MAIN PROGRAM *****/
void main()
{
    unsigned int8   cnt_8ms;            // counter: increments every 8 ms

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


    // Main loop
    while (TRUE)
    {
        // start with both LEDs off
        output_b(0);                    // clear GPIO

        // delay 2 sec
        delay_ms(2000);                 // delay 2000 ms

        // indicate start
        output_high(START);             // turn on start LED

        // wait up to 1 sec for button press
        cnt_8ms = 0;
        while (input(BUTTON) == 1 && cnt_8ms < 1000/8)
        {
            set_timer0(0);                  // clear timer0
            while (get_timer0() < 8000/32)  // wait for 8 ms (32 us/tick)
                ;
            ++cnt_8ms;                      // increment 8 ms counter
        }
        // check elapsed time
        if (cnt_8ms < MAXRT/8)          // if time < max reaction time (8ms/count)
            output_high(SUCCESS);       //   turn on success LED

        // delay 1 sec
        delay_ms(1000);                 // delay 1000 ms

    }   // repeat forever
}
```

Comparisons

As we did in the previous lessons, we can compare, for each language/compiler (MPASM assembler, HI-TECH C and CCS PCB), the length of the source code (ignoring comments and white space) versus program and data memory used by the resulting code:

Reaction_timer

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	53	55	4
HI-TECH C (Lite mode)	23	83	6
CCS PCB	22	84	6

The C source code is around half as long as the assembler version, but the code generated by both C compilers is significantly larger (around 50%) and uses more data memory – again illustrating the trade-off between programmer efficiency (source code length / complexity) and resource-usage.

Example 2: Background Process Timing

We saw in baseline lesson 5 that 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 from example 1, 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
        ; delay 500 ms
        banksel dly_cnt
        movlw   .125            ; repeat 125 times (125 x 4 ms = 500 ms)
        movwf   dly_cnt
dly500  clrf    TMR0            ;   clear timer0
w_tmr0  movf    TMR0,w          ;   wait for 4 ms
        xorlw   .125            ;     (125 ticks x 32 us/tick = 4 ms)
        btfss   STATUS,Z
        goto    w_tmr0
        decfsz  dlycnt,f        ; end 500 ms delay loop
        goto    dly500

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

        ; repeat forever
        goto    main_loop
```

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

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

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

This assembly code, can be implemented in C as:

```
    for (;;)
    {
        // delay 500 ms while responding to button press
        for (dc = 0; dc < 125; dc++)   // repeat 125 times (125 x 4 ms = 500 ms)
        {
            TMR0 = 0;                  //   clear timer0
            while (TMR0 < 125)         //   repeat for 4 ms (125 x 32 us)
            {                          //     check and respond to button press
                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 port (copy shadow to GPIO)
            }
        }
        // toggle flashing LED
        sGPIO ^= 1<<2;                 // toggle LED on GP2 using shadow reg

    }   // repeat forever
```

There is 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.

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 |= 0b000010;          // turn on LED on GP1
            sGPIO &= 0b111101;          // turn off LED on GP1
            sGPIO ^= 0b000100;          // toggle LED on GP2
```

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

If we define symbols representing the port bit positions:

```
#define nFLASH  2               // flashing LED on GP2
#define nPRESS  1               // "button pressed" indicator LED on GP1
```

we can write this statements as:

```
            sGPIO |= 1<<nPRESS;         // turn on indicator LED
            sGPIO &= ~(1<<nPRESS);      // turn off indicator LED
            sGPIO ^= 1<<nFLASH;         // toggle flashing LED
```

These symbols can also be used when configuring the port directions:

```
    TRIS = ~(1<<nFLASH|1<<nPRESS);  // configure LEDs (only) as outputs
```

This makes the code clearer, more general, and therefore more maintainable.

However, this approach doesn't work well on bigger PICs, which have more than one port. You still need to keep track of which port each pin belongs to, and if you change your pin assignments later, you may well need to make a number of changes throughout your code.

A more robust approach is to make use of *bitfields* within C structures.

For example:

```
struct {
    unsigned    GP0     : 1;
    unsigned    GP1     : 1;
    unsigned    GP2     : 1;
    unsigned    GP3     : 1;
    unsigned    GP4     : 1;
    unsigned    GP5     : 1;
} sGPIObits;
```

It is then possible to refer to each bit as a structure member, for example:

```
        sGPIObits.GP1 = 1;
```

and if we also defined a symbol such as:

```
#define sPRESS  sGPIObits.GP1
```

we can then write this as:

```
        sPRESS = 1;
```

That's nice – we have "shadow bits" and we can refer to them easily by symbolic names – but there's still a problem. As well as being able to access individual bits, we also need to be able to refer to the whole shadow register as a single variable, to read or update all the bits at once. After all, that's the whole point of using a shadow register.

We want to be able to change a single bit, as in:

```
        sGPIObits.GP1 = 1;   // set shadow GP1
```

and also read the whole shadow register in a single operation, as in:

```
        GPIO = sGPIO;        // copy shadow register to port
```

How can we do both?

The C *union* construct is intended for exactly this situation, where we need to access the memory holding a variable in more than one way.

We can define for example:

```
union {                             // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;
```

This allows us to refer to the shadow register as `sGPIO.port`, representing the whole port, in a single operation. For example:

```
        sGPIO.port = 0;      // clear shadow register
```

or

```
        GPIO = sGPIO.port;   // update port (copy shadow to GPIO)
```

We can also refer to the individual shadow bits as, for example:

```
        sGPIO.GP1 = 1;       // set shadow GP1
```

If we define symbols representing these shadow bits:

```
#define sFLASH  sGPIO.GP2       // flashing LED (shadow)
#define sPRESS  sGPIO.GP1       // "button pressed" indicator LED (shadow)
```

we can rewrite the previous bit-manipulation statements as:

```
        sPRESS = 1;             // turn on indicator LED
        sPRESS = 0;             // turn off indicator LED
        sFLASH = !sFLASH;       // toggle flashing LED
```

The big advantage of this technique is that, if (on a larger PIC) you were to move one of these functions (such as the flashing LED) to another port, you only need to modify the symbol definition and perhaps your initialisation routine. The rest of your program could stay the same – these statements would still work.

Defining the shadow register as a union incorporating a bitfield structure may seem like a lot of trouble for an apparently small benefit, but it's an elegant approach that will pay off as your applications become more complex.

Complete program

Here is how this shadow register union / bitfield structure definition is used practice:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 3, example 2b                                *
*                                                                       *
*   Demonstates use of Timer0 to maintain timing of background actions  *
*   while performing other actions in response to changing inputs       *
*                                                                       *
*   One LED simply flashes at 1 Hz (50% duty cycle).                    *
*   The other LED is only lit when the pushbutton is pressed            *
*                                                                       *
*   Uses union / bitfield structure to represent shadow register        *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       GP1 = "button pressed" indicator LED                            *
*       GP2 = flashing LED                                              *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <htc.h>
#include <stdint.h>
```

```
/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);

// Pin assignments
#define sFLASH  sGPIO.GP2       // flashing LED (shadow)
#define sPRESS  sGPIO.GP1       // "button pressed" indicator LED (shadow)

#define BUTTON  GP3             // pushbutton


/***** GLOBAL VARIABLES *****/
union {                             // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    uint8_t   dc;                   // delay counter

    // Initialisation
    GPIO = 0;                       // start with all LEDs off
    sGPIO.port = 0;                 //   update shadow
    TRIS = ~(1<<1|1<<2);            // configure GP1 and GP2 (only) 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 (;;)
    {
        // delay 500 ms while responding to button press
        for (dc = 0; dc < 125; dc++)  // repeat 125 times (125 x 4 ms = 500 ms)
        {
            TMR0 = 0;                 //   clear timer0
            while (TMR0 < 125)        //   repeat for 4 ms (125 x 32 us)
            {                         //     check and respond to button press
                sPRESS = 0;           //       assume button up -> indicator off
                if (BUTTON == 0)      //       if button pressed (low)
                    sPRESS = 1;       //         turn on indicator LED (shadow)
                GPIO = sGPIO.port;    //     update port (copy shadow to GPIO)
            }
        }
        // toggle flashing LED
        sFLASH = !sFLASH;             // toggle flashing LED (shadow)

    }   // repeat forever
}
```

CCS PCB

There are no new features to introduce. We only need to convert the references to GPIO and TMR0 in the HI-TECH C code into the CCS PCB built-in function equivalents:

```
    while (TRUE)
    {
        // delay 500 ms while responding to 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)
            {                           //     check and respond to button press
                sPRESS = 0;             //       assume button up -> LED off
                if (input(BUTTON) == 0) //       if button pressed (low)
                    sPRESS = 1;         //         turn on LED (shadow)
                output_b(sGPIO.port);   //     update port (copy shadow to GPIO)
            }
        }
        // toggle flashing LED
        sFLASH = !sFLASH;               // toggle flashing LED (shadow)

    }   // repeat forever
```

The shadow register union is defined in much the same way as for HI-TECH C:

```
union {                             // shadow copy of GPIO
    unsigned int8   port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;
```

and symbols defined to represent the shadow register bits corresponding to the LEDs and pushbutton:

```
#define sFLASH  sGPIO.GP2       // flashing LED (shadow)
#define sPRESS  sGPIO.GP1       // "button pressed" indicator LED (shadow)
#define BUTTON  PIN_B3          // pushbutton on GP3
```

Note that we have to use '`PIN_B3`', instead of '`GP3`'. In previous examples, we defined '`GP3`' as an alias for '`PIN_B3`', but we can't do that anymore, because '`GP3`' has been defined as an element of `sGPIO`.

Comparisons

Here is the resource usage summary for the "Flash an LED while responding to a pushbutton" programs (the C versions defining the shadow register as a union containing a bitfield structure, as above):

Flash+PB_LED

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	37	31	2
HI-TECH C (Lite mode)	30	67	4
CCS PCB	29	47	6

The C source code is comparatively long in this example, because of the shadow register union / bitfield structure definition. It's a big part of the source code – something you wouldn't normally bother with, for such a small program. But we'll keep doing it this way, because it's good practice that will serve us well as our programs become longer, and the extra lines of variable definition won't seem to be such a big deal.

It's interesting to note that the code generated by HI-TECH C (in 'Lite' mode) is now much larger than that generated by the CCS compiler – showing the impact of having most optimisation disabled. The paid-for versions of HI-TECH C could be expected to generate more compact code.

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 is perfectly reasonable to use it to debounce switches.

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

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 10 ms (157 x 64 us = 10 ms)
            if (GP3 == 1)           //   if button up,
                TMR0 = 0;           //     restart wait
```

Using C macros

This fragment of code is one that we might want to use a number of times, perhaps modified to debounce switches on inputs other than GP3, in this or other programs.

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 can also be used when programming in C.

For example, we could define our debounce routine as a macro as follows:

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

// DbnceLo()
//
// Debounce switch on given input pin
// Waits for switch input to be low continuously for DEBOUNCE*256/1000 ms
//
// Uses: TMR0       Assumes: TMR0 running at 256 us/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 used within your program as, for example:

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

You can define macros toward the start of your source code, but as you build your own library of useful macros, you would normally keep them together in one or more header files, such as 'stdmacros.h', and reference them from your main program, using the `#include` directive.

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 then released                *
*                                                                       *
************************************************************************
*   Pin assignments:                                                    *
*       GP1 = flashing LED                                              *
*       GP3 = pushbutton switch (active low)                            *
*                                                                       *
************************************************************************/

#include <htc.h>
#include <stdint.h>


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_OFF & CP_OFF & WDT_OFF & OSC_IntRC);

// Pin assignments
#define sFLASH  sGPIO.GP1           // flashing LED (shadow)
#define BUTTON  GP3                 // pushbutton


/***** GLOBAL VARIABLES *****/
union {                             // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure port
    GPIO = 0;                       // start with LED off
    sGPIO.port = 0;                 //   update shadow
    TRIS = 0b111101;                // configure GP1 (only) as an output

    // configure timer
    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 for button press, debounce using timer0:
        TMR0 = 0;                   // reset timer
        while (TMR0 < 157)          // wait at least 10 ms (157 x 64 us = 10 ms)
            if (BUTTON == 1)        //   if button up,
                TMR0 = 0;           //     restart wait

        // toggle LED
        sFLASH = !sFLASH;           // toggle flashing LED (shadow)
        GPIO = sGPIO.port;          // write to GPIO

        // wait for button release, debounce using timer0:
        TMR0 = 0;                   // reset timer
        while (TMR0 < 157)          // wait at least 10ms (157 x 64us = 10ms)
            if (BUTTON == 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 10 ms (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 = 10 ms/(256us/tick)

// DbnceLo()
//
// Debounce switch on given input pin
// Waits for switch input to be low continuously for 10 ms
//
// Uses: TMR0       Assumes: TMR0 running at 256 us/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:

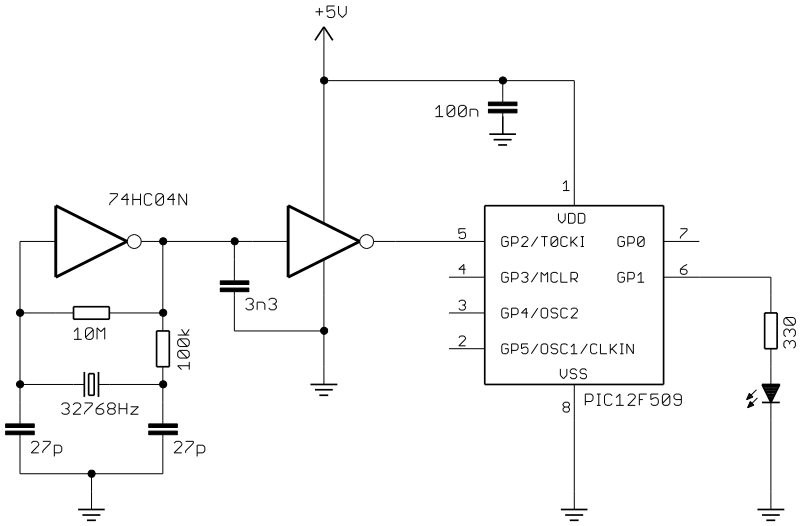
```
/************************************************************************
*                                                                       *
*   Description:    Lesson 2, example 3a                                *
*                                                                       *
*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

#include <12F509.h>


/***** CONFIGURATION *****/
// int reset, no code protect, no watchdog, int RC clock
#fuses NOMCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define sFLASH  sGPIO.GP1       // flashing LED (shadow)
#define BUTTON  PIN_B3          // pushbutton on GP3


/***** GLOBAL VARIABLES *****/
union {                             // shadow copy of GPIO
    unsigned int8   port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure port
    output_b(0);                    // start with LED off
    sGPIO.port = 0;                 //   update shadow

    // configure Timer0:
    setup_timer_0(RTCC_INTERNAL|RTCC_DIV_64);   // timer mode, prescale = 64
                                                // -> increment every 64 us

    // Main loop
    while (TRUE)
```

```
    {
        // wait for button press, debounce using timer0:
        set_timer0(0);                  // reset timer
        while (get_timer0() < 157)      // wait at least 10ms (157x64us = 10ms)
            if (input(BUTTON) == 1)     //   if button up,
                set_timer0(0);          //     restart wait

        // toggle LED
        sFLASH = !sFLASH;               // toggle flashing LED (shadow)
        output_b(sGPIO.port);           // 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 (157x64us = 10ms)
            if (input(BUTTON) == 0)     //   if button down,
                set_timer0(0);          //     restart wait

    }   // repeat forever
}
```

Example 4: Using Counter Mode

So far we've used Timer0 in "timer mode", where it is clocked by the PIC's instruction clock, which runs at one quarter the speed of the processor clock (i.e. 1 MHz when the 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. An external 32.768 kHz crystal oscillator (as shown on the right) is used to drive the counter, providing a time base that can be used to flash an LED at a more accurate 1 Hz.



To configure the Gooligum baseline training board for use with this example, connect jumpers JP22 (connecting the 32 kHz clock signal to T0CKI) and JP12 (enabling the LED on GP1).
If you are using Microchip's Low Pin Count Demo Board, you will need to build the oscillator circuit separately, as described in baseline lesson 5.

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 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:

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

        movf    sGPIO,w         ; copy shadow to GPIO
        movwf   GPIO

        ; repeat forever
        goto    loop
```

HI-TECH C

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
            sFLASH = 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 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>
#include <stdint.h>


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
__CONFIG(MCLRE_ON & CP_OFF & WDT_OFF & OSC_IntRC);
```

```
// Pin assignments
#define sFLASH  sGPIO.GP1           // flashing LED (shadow)


/***** GLOBAL VARIABLES *****/
union {                             // shadow copy of GPIO
    uint8_t         port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

    // configure port
    TRIS = 0b111101;            // configure GP1 (only) as an output

    // configure timer
    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 1 Hz, so continually copy to LED
        sFLASH = 0;                 // assume TMR<7>=0 -> LED off
        if (TMR0 & 1<<7)            // if TMR0<7>=1
            sFLASH = 1;             //   turn on LED

        GPIO = sGPIO.port;          // 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`'.

In this example, we don't care if the counter increments on rising or falling edges – it will count at the same rate in either case. So it doesn't matter whether we use '`RTCC_EXT_L_TO_H`' or '`RTCC_EXT_H_TO_L`' here.

We can configure the timer with either:

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

or

```
    setup_timer_0(RTCC_EXT_H_TO_L|RTCC_DIV_128);
```

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

```
if (get_timer0() & 1<<7)     // if TMR0<7>=1
    sFLASH = 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
    sFLASH = 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 discourages 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


/***** CONFIGURATION *****/
// ext reset, no code protect, no watchdog, int RC clock
#fuses MCLR,NOPROTECT,NOWDT,INTRC

// Pin assignments
#define sFLASH  sGPIO.GP1           // flashing LED (shadow)


/***** GLOBAL VARIABLES *****/
union {                             // shadow copy of GPIO
    unsigned int8   port;
    struct {
        unsigned    GP0     : 1;
        unsigned    GP1     : 1;
        unsigned    GP2     : 1;
        unsigned    GP3     : 1;
        unsigned    GP4     : 1;
        unsigned    GP5     : 1;
    };
} sGPIO;


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation

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

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

        output_b(sGPIO.port);       // copy shadow to GPIO

    }   // repeat forever
}
```

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 being able to be expressed quite succinctly in C.

We've also seen that using symbolic names and macros can help make your code more maintainable, and how the union and bitfield structure constructs can be used to make it possible to access both a whole variable and its individual bits, in an elegant way.

In the next lesson we'll see how these C compilers can be used with processor features such as sleep mode and the watchdog timer, and to select various clock, or oscillator, configurations.