Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 9: Using Timer1

Midrange assembler lesson 15 introduced the 16-bit timer/counter, Timer1, showing how it can be used for everything that Timer0 can do, and more, such as continuing to run while the device is in sleep mode, and (in most midrange PICs) being capable of being gated by an external digital signal or comparator output.

This lesson revisits that material, showing how to use C to control and access Timer1, re-implementing the examples using the free HI-TECH C[1] (in "Lite" mode) and PICC-Lite compilers.

In summary, this lesson covers:

- Introduction to the Timer1 module
- Using Timer1 to drive an interrupt
- Using Timer1 with an external 32.768 kHz crystal
- Operating Timer1 in sleep mode
- Measuring pulse widths by gating Timer1, using a digital input or comparator output.

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

PIC16F684 Timer1 Module[2]

Timer1 consists of a pair of 8-bit registers: TMR1H and TMR1L, holding the most and least significant bytes of the 16-bit timer.

As with all the other special function registers, the HI-TECH C compilers make these available as two `unsigned char` (8-bit) variables, `TMR1H` and `TMR1L`.

HI-TECH C version 9.81 provides include files which also define `TMR1` as an `unsigned int` (16-bit) variable, which can be used to access all 16 bits of the timer "at once". Of course, given that the PIC is an 8-bit device, the compiler has to implement each "16-bit" access as two separate 8-bit operations. As we saw in midrange assembler lesson 15, this means that reading or writing a 16-bit timer while it is incrementing can be problematic. In general, it is easiest to simply stop Timer1 before accessing it.

Timer1 is configured using the T1CON register.

The TMR1ON bit is used to start and stop the timer: to enable Timer1, set TMR1ON.

---

[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.microchip.com.
[2] This section describes the Timer1 module found in most midrange PICs. Some newer midrange devices include a (very similar) enhanced Timer1 module, with some additional options.

TMR1CS selects the timer's clock source:

'0' selects timer mode, where TMR1 is incremented at a fixed rate by the instruction clock (FOSC/4)

'1' selects counter mode, where TMR1 is incremented by an external signal, on the T1CKI pin.

When in counter mode, Timer1 can run *asynchronously*, where TMR1 increments independently of the PIC's internal clocks, and is not synchronised with the PIC's instruction execution.

The counter mode is set by the $\overline{\text{T1SYNC}}$ bit[3]:

'0' selects synchronous mode

'1' selects asynchronous mode (where the counter is not synchronised with the PIC's internal clocks)

The T1OSCEN bit is used to enable the timer's 32.768 kHz crystal oscillator, and the TMR1GE and T1GINV bits control the timer's *gate* facility, described later in this lesson.

The timer's prescale ratio is set by the T1CKPS<1:0> bits, as shown in the following table:

T1CKPS<1:0> bit value	Timer1 prescale ratio
00	1 : 1
01	1 : 2
10	1 : 4
11	1 : 8

T1CKPS<1:0> = '00' means that no prescaling will occur, and TMR1 will increment at the input clock or instruction cycle rate.

T1CKPS<1:0> = '11' selects the maximum prescale ratio of 1:8, meaning that TMR1 will increment every 8 instruction cycles in timer mode. Given a 1 MHz instruction cycle rate, the timer would increment every 8 µs.

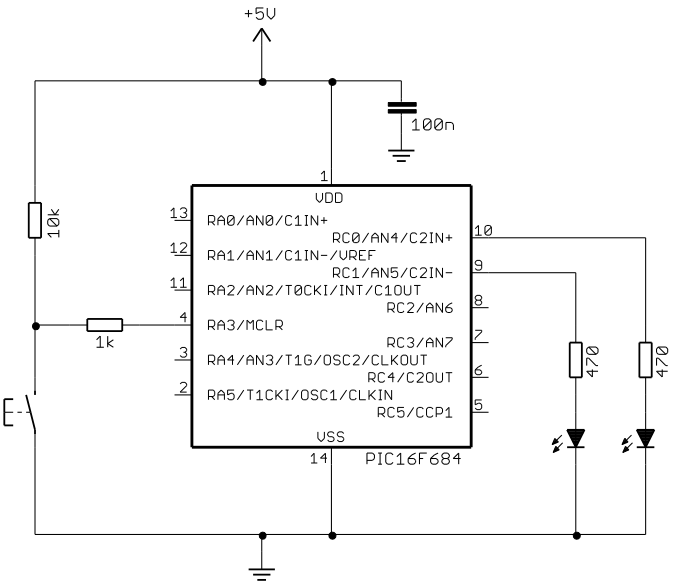
Example 1: Flash LED while responding to input

Lesson 2 included an example where an LED was flashed at a steady rate (approx 1 Hz), while another LED was lit in response to a pushbutton press.

In this example, we will do the same thing, using Timer1.

We will use the circuit shown on the right, based on a PIC16F684 with LEDs on RC0 and RC1. If you are using Microchip's Low Pin Count Demo Board, you'll already have this circuit – just plug in the PIC16F684!

The LED on RC1 will flash at approx 1 Hz, and the LED on RC0 will be lit only while the pushbutton on RA3 is pressed.



Given a 4 MHz processor clock and using a 1:8 prescale ratio, Timer1 can time events up to 524.28 ms. So, to generate a 500 ms

[3] The include files provided with PICC-Lite and HI-TECH C up to version 9.80 defined this bit as `T1SYNC`. However, the include files provided with HI-TECH C v9.81 define it as `nT1SYNC`.

delay, we can simply load TMR1 with an appropriate initial value, to make it overflow after 500 ms.

Given a 4 MHz processor clock and a 1:8 prescale ratio, Timer1 will increment every 8 µs. Thus, for a delay of 500 ms, we need 500000 ÷ 8 = 62500 ticks.

For TMR1 to overflow after 62500 counts, we need to load it with an initial value of 65536 – 62500 = 3036.

Instead of using the value '3036' directly, the code will be easier to maintain if we define the delay period as a constant, and allow the compiler to calculate the initial value:

```
#define FlashMS 500                     // LED flash toggle time in milliseconds
#define InitT1  65536-FlashMS*1000/8    // Initial value to load into TMR1
                                        //   to generate FlashMS delay
                                        //   (assuming 8 us/tick)
```

Timer1 is initialised, and the overflow (interrupt) flag cleared, as follows:

```
    T1CON = 0b00110000;             // configure Timer1:
            //-0------                   gate disabled (TMR1GE = 0)
            //--11----                   prescale = 8 (T1CKPS = 11)
            //----0---                   LP oscillator disabled (T1OSCEN = 0)
            //------0-                   internal clock (TMR1CS = 0)
            //-------0                   disable Timer1 (TMR1ON = 0)
                                    //  -> increment TMR1 every 8 us
    TMR1IF = 0;                     //  clear overflow flag
```

At the start of the main loop, we need to load the 16-bit value, defined by '`InitT1`' (and equal to 3036 in this case), into TMR1.

As mentioned above, it is best to stop Timer1, load the value, and then restart it.

This can be done by using bitwise logical operators to extract the high and low bytes of the initial value, and writing them to TMR1H and TMR1L respectively:

```
        TMR1ON = 0;                 // stop Timer1
        TMR1H = InitT1 >> 8;        // load TMR1 with initial value (InitT1)
        TMR1L = InitT1 & 0xFF;      //   to generate 500 ms count
        TMR1ON = 1;                 // start Timer1
```

Or, if using HI-TECH C version 9.81, it is possible to write this as single 16-bit assignment:

```
        TMR1ON = 0;                 // stop Timer1
        TMR1 = InitT1;              // load TMR1 with initial value (InitT1)
                                    //   to generate 500 ms count
        TMR1ON = 1;                 // start Timer1
```

Note again that, even though '`TMR1 = InitT1`' looks like a single operation, the compiler has to implement this 16-bit assignment as two separate 8-bit operations, meaning that it is still good practice to stop Timer1 before writing to it, as shown.

We can then wait until Timer1's overflow flag is set, while monitoring and responding to button presses:

```
        while (!TMR1IF)             // repeat until Timer1 overflows
        {
            // respond to button press
            if (!BUTTON)                    // if button pressed (low)
                sPORTC |= 1<<nB_LED;        //   turn on indicator LED
            else                            // else
                sPORTC &= ~(1<<nB_LED);     //   turn it off
```

```
            // copy shadow register to port
            PORTC = sPORTC;
        }
```

At this point, TMR1IF is set, indicating that TMR1 has overflowed and that therefore 500 ms has elapsed.

We can now clear the interrupt flag and toggle the flashing LED:

```
        TMR1IF = 0;                     // clear overflow flag for next time

        // flash LED
        sPORTC ^= 1<<nF_LED;
```

and then repeat.

Complete program

Here's the complete Timer1 version of the flash + pushbutton example, for HI-TECH C v9.81:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 9, example 1                                 *
*                                                                       *
*   Demonstrates use of Timer1 to maintain timing of background actions *
*   while performing other actions in response to changing inputs       *
*                                                                       *
*   One LED simply flashes at 1 Hz (50% duty cycle).                    *
*   The other LED is only lit when the pushbutton is pressed            *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RC0 - "button pressed" indicator LED                            *
*       RC1 - flashing LED                                              *
*       RA3 - pushbutton (active low)                                   *
*                                                                       *
************************************************************************/

#include <htc.h>


/***** CONFIGURATION *****/
//  int reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF & PWRTE_ON &
         FOSC_INTOSCIO & FCMEN_OFF & IESO_OFF);

// Pin assignments
#define nB_LED  0           // "button pressed" indicator LED on RC0
#define nF_LED  1           // flashing LED on RC1
#define BUTTON  RA3         // pushbutton (active low)


/***** CONSTANTS *****/
#define FlashMS 500                     // LED flash toggle time in milliseconds
#define InitT1  65536-FlashMS*1000/8    // Initial value to load into TMR1
                                        //   to generate FlashMS delay
                                        //   (assuming 8 us/tick)
```

```
/***** GLOBAL VARIABLES *****/
unsigned char   sPORTC;                 // shadow copy of PORTC


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

    // configure ports
    PORTC = 0;                          // start with PORTC clear (all LEDs off)
    sPORTC = 0;                         //   and update shadow
    TRISC = 0;                          // configure all PORTC pins as outputs

    // configure timers
    T1CON = 0b00110000;                 // configure Timer1:
            //-0------                      gate disabled (TMR1GE = 0)
            //--11----                      prescale = 8 (T1CKPS = 11)
            //----0---                      LP oscillator disabled (T1OSCEN = 0)
            //------0-                      internal clock (TMR1CS = 0)
            //-------0                      disable Timer1 (TMR1ON = 0)
                                        //  -> increment TMR1 every 8 us
    TMR1IF = 0;                         //  clear overflow flag


    // Main loop
    for (;;)
    {
        // delay ~500ms while checking for button press
        TMR1ON = 0;                     // stop Timer1
        TMR1 = InitT1;                  // load TMR1 with initial value (InitT1)
                                        //   to generate 500 ms count
        TMR1ON = 1;                     // start Timer1

        while (!TMR1IF)                 // repeat until Timer1 overflows
        {
            // respond to button press
            if (!BUTTON)                        // if button pressed (low)
                sPORTC |= 1<<nB_LED;            //   turn on indicator LED
            else                                // else
                sPORTC &= ~(1<<nB_LED);         //   turn it off

            // copy shadow register to port
            PORTC = sPORTC;
        }
        // (end 500 ms delay)

        TMR1IF = 0;                         // clear overflow flag for next time

        // flash LED
        sPORTC ^= 1<<nF_LED;
    }
}
```

Example 2: Using an interrupt to flash LED while responding to input

In lesson 3, we saw how Timer0 can be used to drive a regular interrupt which can be used to perform "background" tasks, such as flashing an LED, independently of the rest of the code. Timer1 can be used to

generate a regular interrupt in much the same way. To demonstrate how it's done, we'll re-implement the first example, using a Timer1 interrupt.

Timer1 interrupts

We've already seen that, whenever TMR1 overflows, the TMR1IF flag in the PIR1 register is set. This is equivalent to Timer0's T0IF interrupt flag.

As with other interrupt sources, an interrupt will be triggered whenever the TMR1IF flag is set, if the Timer1 interrupt is enabled by setting the TMR1IE enable bit is in the PIE1 register.

And like the other peripherals, such as the comparators and ADC, to enable Timer1 interrupts you must also set the peripheral and global interrupt enable bits (PEIE and GIE) in the INTCON register.

So, to enable the Timer1 interrupt, our Timer1 initialisation code now becomes:

```
// configure timers
T1CON = 0b00110000;             // configure Timer1:
        //-0------                   gate disabled (TMR1GE = 0)
        //--11----                   prescale = 8 (T1CKPS = 11)
        //----0---                   LP oscillator disabled (T1OSCEN = 0)
        //------0-                   internal clock (TMR1CS = 0)
        //-------0                   disable Timer1 (TMR1ON = 0)
                                //  -> increment TMR1 every 8 us
TMR1H = InitT1 >> 8;            // load TMR1 with initial value (InitT1)
TMR1L = InitT1 & 0xFF;          //   to overflow after 500 ms
TMR1ON = 1;                     // start Timer1
TMR1IE = 1;                     // enable Timer1 interrupt

// enable interrupts
PEIE = 1;                       // enable peripheral
ei();                           //   and global interrupts
```

At the start of the ISR, we must clear the interrupt flag, as usual:

```
TMR1IF = 0;                     // clear interrupt flag
```

As explained in lesson 3, for greater timing accuracy, it is better to add an offset to the current value of the timer, than to load a new initial value, because the timer will have continued to count, from the time the interrupt was triggered (when the timer overflowed) until the Timer1 interrupt handler code is run. This is especially likely if other interrupt sources are active; if TMR1 overflows while another interrupt handler is active it may be some time before the Timer1 interrupt is serviced. By adding a fixed offset to the current count, whatever it may be, the total interval between Timer1 interrupts will be constant.

To perform the 16-bit addition, it's easiest to simply stop the timer first.

For PICC-Lite, we have:

```
// add offset to Timer1 for overflow after 500 ms
TMR1ON = 0;                     // stop Timer1
TMR1L += (InitT1+1) & 0xFF;     // add 16-bit offset (initial value
                                //  adjusted for addition overhead) to TMR1
if (TMR1L < ((InitT1+1) & 0xFF))    // (handle lower-byte overflow)
    ++TMR1H;
TMR1H += (InitT1+1) >> 8;       // to generate ~500 ms count
TMR1ON = 1;                     // start Timer1
```

The addition is done in two parts, because we can only access the timer's high and low bytes separately.

After adding to TMR1L, we need to test to see whether the addition overflowed, and if so, to increment TMR1H. In assembler, we use the carry flag for this, and indeed with HI-TECH C we could have used:

```
TMR1L += (InitT1+1) & 0xFF;      // add 16-bit offset (initial value
                                 //  adjusted for addition overhead) to TMR1
if (CARRY)                       //    (handle lower-byte overflow)
    ++TMR1H;
```

In practice, this will work. But in principle, we cannot guarantee what PIC instructions the C compiler will generate, and that this will not change between compiler releases. We should not assume that the compiled code will always affect the carry, or any other status flag, in the way that we expect.

But we can be sure that the result of an 8-bit addition, truncated to 8 bits (using a logical AND), will always be at least as large as the amount being added, unless the addition overflowed, which is why the overflow test was written as:

```
TMR1L += (InitT1+1) & 0xFF;      // add 16-bit offset (initial value
                                 //  adjusted for addition overhead) to TMR1
if (TMR1L < ((InitT1+1) & 0xFF))    // (handle lower-byte overflow)
    ++TMR1H;
```

With HI-TECH C version 9.81, we can access TMR1 as a single 16-bit quantity, making it possible to simply write:

```
// add offset to Timer1 for overflow after 500 ms
TMR1ON = 0;                      // stop Timer1
TMR1 += InitT1+1;                // add 16-bit offset (initial value
                                 //  adjusted for addition overhead) to TMR1
                                 //  to generate ~500 ms count
TMR1ON = 1;                      // start Timer1
```

That's certainly a lot clearer!

Note the offset being added is increased by one timer tick (equivalent to eight instruction cycles), to account for the fact that time elapses while the timer is stopped, while the addition is taking place. This cannot be completely accurate, because we cannot know exactly how many cycles the code generated by the compiler will take to perform the addition. You could look at the disassembly listing for the compiled code, or use the MPLAB simulator to measure the delay, but you cannot guarantee that this timing will remain the same with future compiler releases.

If you wanted a cycle-exact 500 ms delay, you could turn off the prescaler and take the approach used in lesson 3, where Timer1 would be set to overflow every (say) 50 ms, and a variable to used as a counter, so that the LED is toggled every 10 interrupts, i.e. 500 ms. And we'll other ways[4] to do this, in future lessons.

Having adjusted Timer1 so that it will overflow in another 500 ms (or so), we can toggle the LED:

```
// flash LED
sPORTC ^= 1<<nF_LED;
```

before exiting the ISR.

[4] such as using the CCPR1H:CPPR1L register pair in the ECCP module as a period register for TMR1

The main loop now only needs to respond to button presses and update the port from its shadow register (which may have been modified by the ISR):

```
    // Main loop
    for (;;)
    {
        // respond to button press
        if (!BUTTON)                    // if button pressed (low)
            sPORTC |= 1<<nB_LED;        //   turn on indicator LED
        else                            // else
            sPORTC &= ~(1<<nB_LED);     //   turn it off

        // copy shadow register (updated by ISR) to port
        PORTC = sPORTC;
    }
```

Complete program

Here's how it all fits together, using HI-TECH C v9.81:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 9, example 2                                 *
*                                                                       *
*   Demonstrates use of Timer1 interrupt to perform a background task   *
*   while main loop performs other actions                              *
*                                                                       *
*   One LED flashes at approx 1 Hz (50% duty cycle).                    *
*   The other LED is only lit when the pushbutton is pressed            *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RC0 - "button pressed" indicator LED                            *
*       RC1 - flashing LED                                              *
*       RA3 - pushbutton (active low)                                   *
*                                                                       *
************************************************************************/

#include <htc.h>


/***** CONFIGURATION *****/
//  int reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLRE_OFF & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF & PWRTE_ON &
         FOSC_INTOSCIO & FCMEN_OFF & IESO_OFF);

// Pin assignments
#define nB_LED  0           // "button pressed" indicator LED on RC0
#define nF_LED  1           // flashing LED on RC1
#define BUTTON  RA3         // pushbutton (active low)


/***** CONSTANTS *****/
#define FlashMS 500                     // LED flash toggle time in milliseconds
#define InitT1  65536-FlashMS*1000/8    // Initial value to load into TMR1
                                        //   to generate FlashMS delay
                                        //   (assuming 8 us/tick)
```

```
/***** GLOBAL VARIABLES *****/
volatile unsigned char  sPORTC;         // shadow copy of PORTC


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

    // configure ports
    PORTC = 0;                      // start with PORTC clear (all LEDs off)
    sPORTC = 0;                     //   and update shadow
    TRISC = 0;                      // configure all PORTC pins as outputs

    // configure timers
    T1CON = 0b00110000;             // configure Timer1:
            //-0------                   gate disabled (TMR1GE = 0)
            //--11----                   prescale = 8 (T1CKPS = 11)
            //----0---                   LP oscillator disabled (T1OSCEN = 0)
            //------0-                   internal clock (TMR1CS = 0)
            //-------0                   disable Timer1 (TMR1ON = 0)
                                    //  -> increment TMR1 every 8 us
    TMR1 = InitT1;                  // load TMR1 with initial value (InitT1)
                                    //   to overflow after 500 ms
    TMR1ON = 1;                     // start Timer1
    TMR1IE = 1;                     // enable Timer1 interrupt

    // enable interrupts
    PEIE = 1;                       // enable peripheral
    ei();                           //   and global interrupts


    // Main loop
    for (;;)
    {
        // respond to button press
        if (!BUTTON)                    // if button pressed (low)
            sPORTC |= 1<<nB_LED;        //   turn on indicator LED
        else                            // else
            sPORTC &= ~(1<<nB_LED);     //   turn it off

        // copy shadow register (updated by ISR) to port
        PORTC = sPORTC;
    }
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    // *** Service Timer1 interrupt
    //  TMR1 overflows every 500 ms (approx)
    //
    //  Flashes LED at ~1 Hz by toggling on each interrupt
    //      (every ~500 ms)
    //
    //  (only Timer1 interrupts are enabled)
    //
    TMR1IF = 0;                     // clear interrupt flag
```

```
    // add offset to Timer1 for overflow after 500 ms
    TMR1ON = 0;                     // stop Timer1
    TMR1 += InitT1+1;               // add 16-bit offset (initial value
                                    //  adjusted for addition overhead) to TMR1
                                    //  to generate ~500 ms count
    TMR1ON = 1;                     // start Timer1

    // flash LED
    sPORTC ^= 1<<nF_LED;
}
```

Timer1 Oscillator

32.768 kHz "watch" crystals are commonly used for time-keeping, because they are low cost but provide a reasonably stable and precise oscillation which is easily divided to give a time-base in seconds – very useful when implementing a real-time clock (RTC), within an embedded application, or even a standalone clock, such as the Gooligum travel clock kit.

Timer1 provides direct support for these watch crystals, through the PIC's LP (low power) crystal oscillator, while the internal oscillator is used as the system clock. This means that an external clock circuit; the 32.768 kHz crystal can be driven directly by the OSC1 and OSC2 pins, as shown in the next example.

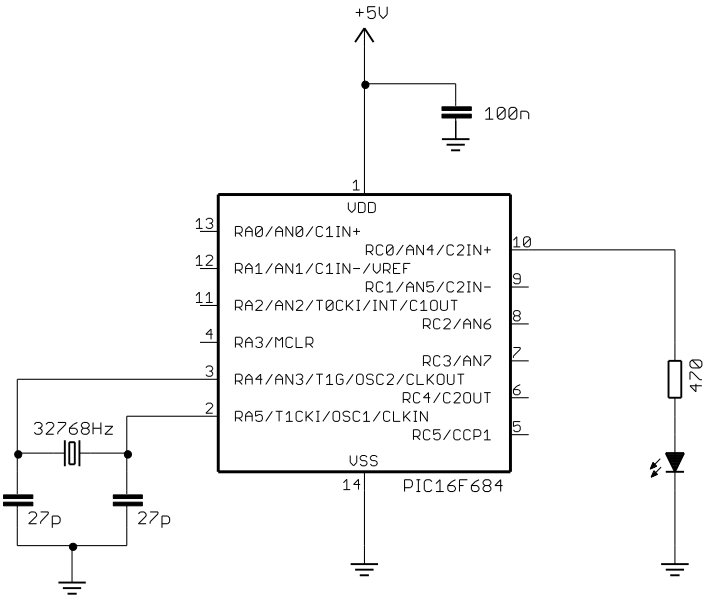
To enable the LP oscillator for use by Timer1, set the T1OSCEN bit in T1CON.

To use this oscillator to drive Timer1, the TMR1CS bit must also be set. This selects counter mode, where TMR1 is driven by a signal on T1CKI[5], which shares its pin with OSC1.

Example 3: Timer1 Oscillator

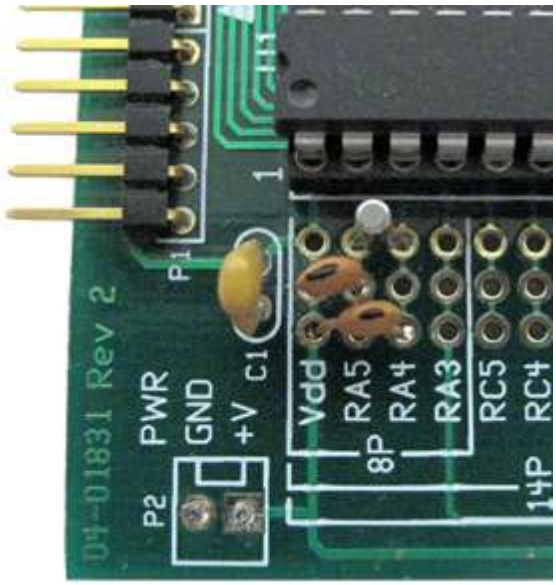
To show how easy it is to use a 32.768 kHz crystal with Timer1, we will re-implement the counter example from lesson 2. But instead of the external clock circuit driving Timer0, we'll use Timer1's LP oscillator, resulting in a much simpler circuit.

To do this, connect the watch crystal between the OSC1 and OSC2 pins, with 27 pF (22 pF or 33 pF will also work) loading capacitors between the crystal and ground, as shown on the right.



It is possible to build this circuit by connecting a breadboard to the 14-pin header on the Microchip Low Pin Count Demo Board, but doing so may lead to reliability issues. You will have fewer problems if you solder the crystal and capacitors directly to the LPC Demo Board, as illustrated on the next page.

[5] Like T0CKI, the T1CKI pin is a Schmitt Trigger digital input.



Note that the capacitors in the picture are **not** connected as per the circuit diagram! For convenience (the layout is a lot simpler), they are connected between the crystal and V_{DD} (+5V), not ground. This works because capacitors block DC; as far as the crystal is concerned, its load capacitors are grounded, because the capacitors block the DC component of the 5 V supply.

The software is also quite straightforward.

Firstly, we need to place Timer1 in counter mode, with the LP oscillator enabled and no prescaling:

```
// configure timers
T1CON = 0b00001011;             // configure Timer1:
        //-0------                   gate disabled (TMR1GE = 0)
        //--00----                   prescale = 1 (T1CKPS = 00)
        //----1---                   LP oscillator enabled (T1OSCEN = 1)
        //------1-                   external clock (TMR1CS = 1)
        //-------1                   enable Timer1 (TMR1ON = 1)
                                //  -> increment TMR1 at 32.768 kHz
```

Timer1 will now increment at 32.768 kHz.

At this clock rate, bit 14 of TMR1 cycles once per second.

This is bit 6 of TMR1H, so if we simply copy TMR1H<6> to the pin the LED is connected to, the LED will flash at 1 Hz:

```
// TMR1<14> (= TMR1H<6>) cycles at 1 Hz
// so continually copy to LED
if (TMR1H & 1<<6)                   // if TMR1H<6> = 1
    sPORTC |= 1<<nF_LED;            //   turn on LED
else                                // else
    sPORTC &= ~(1<<nF_LED);         //   turn it off
```

Or, if using HI-TECH C v9.81, we can write the same thing a little more clearly as:

```
// TMR1<14> cycles at 1 Hz
// so continually copy to LED
if (TMR1 & 1<<14)                   // if TMR1<14> = 1
    sPORTC |= 1<<nF_LED;            //   turn on LED
else                                // else
    sPORTC &= ~(1<<nF_LED);         //   turn it off
```

We can then copy the shadow register to PORTC, as normal:

```
PORTC = sPORTC;
```

and repeat.

Complete program

Here is the complete PICC-Lite version of this example:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 9, example 3                                 *
*                                                                       *
*   Demonstrates basic use of Timer1 LP oscillator                      *
*                                                                       *
*   LED flashes at 1 Hz (50% duty cycle), with timing derived           *
*   from 32.768 kHz crystal driven by Timer1 oscillator                 *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RC0        - flashing LED                                       *
*       OSC1, OSC2 - 32.768 kHz crystal                                 *
*                                                                       *
************************************************************************/

#include <htc.h>


/***** CONFIGURATION *****/
//  ext reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLREN & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO & FCMDIS &
         IESODIS);

// Pin assignments
#define nF_LED  0           // flashing LED on RC0


/***** GLOBAL VARIABLES *****/
unsigned char   sPORTC;                 // shadow copy of PORTC


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

    // configure ports
    PORTC = 0;                      // start with PORTC clear (all LEDs off)
    sPORTC = 0;                     //   and update shadow
    TRISC = 0;                      // configure all PORTC pins as outputs

    // configure timers
    T1CON = 0b00001011;             // configure Timer1:
            //-0------                   gate disabled (TMR1GE = 0)
            //--00----                   prescale = 1 (T1CKPS = 00)
            //----1---                   LP oscillator enabled (T1OSCEN = 1)
            //------1-                   external clock (TMR1CS = 1)
            //-------1                   enable Timer1 (TMR1ON = 1)
                                    //  -> increment TMR1 at 32.768 kHz


    // Main loop
    for (;;)
    {
```

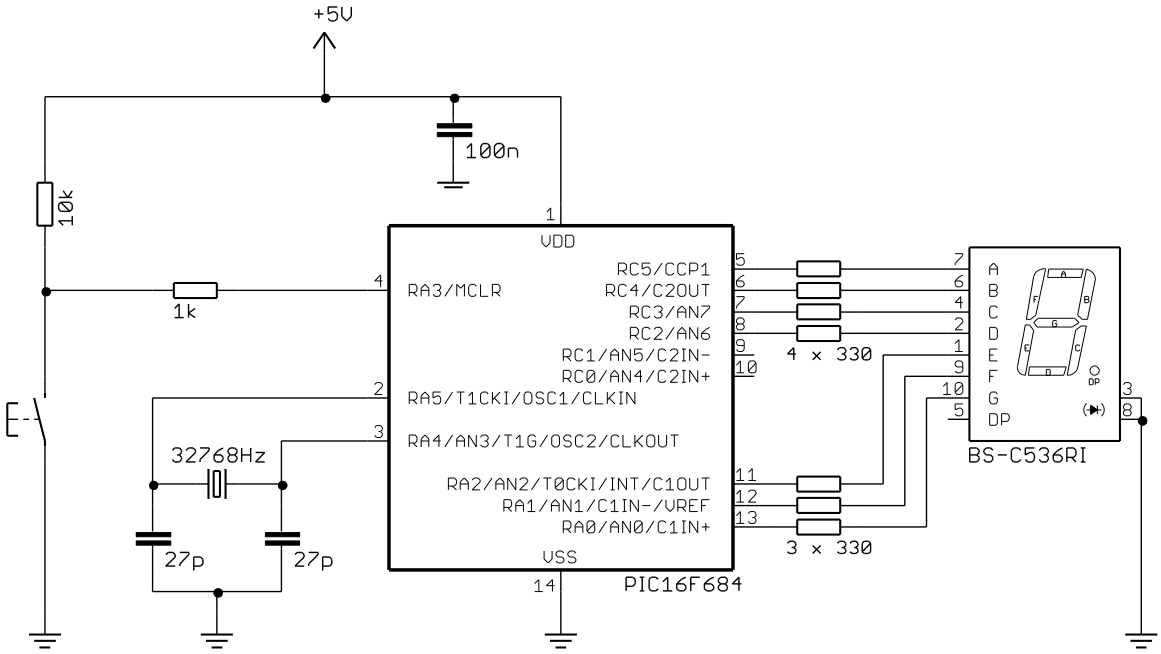
```
        // TMR1<14> (= TMR1H<6>) cycles at 1 Hz
        // so continually copy to LED
        if (TMR1H & 1<<6)                   // if TMR1H<6> = 1
            sPORTC |= 1<<nF_LED;            //   turn on LED
        else                                // else
            sPORTC &= ~(1<<nF_LED);         //   turn it off

        // copy shadow register to port
        PORTC = sPORTC;
    }
}
```

Example 4: Seconds Counter

This example doesn't introduce anything new, but by combining the Timer1 oscillator with Timer1 interrupts and a 7-segment display we'll be set up for the section on sleep mode.

We will implement a single-digit seconds counter, with timing derived from a 32.768 kHz crystal, making use of the Timer1 oscillator and interrupts. This could form the basis for a digital clock or a more elaborate timer – simply extend the display with more digits, using the multiplexing technique introduced in lesson 7.

The circuit is shown below. You'll see that it incorporates elements from previous examples, including the single-digit 7-segment display from lesson 7. The pushbutton isn't used here (except as a reset button; we will enable external $\overline{\text{MCLR}}$), but we'll need it in the next example.



Timer1 is configured as before, driven by the 32.768 kHz crystal:

```
    T1CON = 0b00001010;             // configure Timer1:
            //-0------                   gate disabled (TMR1GE = 0)
            //--00----                   prescale = 1 (T1CKPS = 00)
            //----1---                   LP oscillator enabled (T1OSCEN = 1)
            //------1-                   external clock (TMR1CS = 1)
            //-------0                   disable Timer1 (TMR1ON = 0)
                                    //  -> increment TMR1 at 32.768 kHz
```

We will use a Timer1 interrupt to update the seconds count (in the background), while the main loop handles the display.

To ensure that the Timer1 interrupt does not trigger prematurely (at power-on, the initial value of TMR1 is undefined), we will initialise TMR1 to zero, before enabling the interrupt:

```
TMR1H = 0;                      // clear TMR1 (start counting from 0)
TMR1L = 0;
TMR1ON = 1;                     // start Timer1
TMR1IE = 1;                     // enable Timer1 interrupt
```

Or, if using HI-TECH C version 9.81, you can write:

```
TMR1 = 0;                       // clear TMR1 (start counting from 0)
TMR1ON = 1;                     // start Timer1
TMR1IE = 1;                     // enable Timer1 interrupt
```

We'll need a variable to hold the current seconds count. It will be updated by the Timer1 interrupt handler and accessed in the main loop, so it has to be a global variable, and defined as `volatile`.

It should be initialised to zero, so that our seconds count display starts from zero:

```
/***** GLOBAL VARIABLES *****/
volatile unsigned char  t1_secs = 0;    // seconds count (even only),
                                        //  updated by Timer1 interrupt
```

The Timer1 interrupt is triggered every 2 s, when TMR1 overflows (a 16-bit counter, incremented at 32768 Hz, will overflow after 65536 counts = 2 seconds).

You can see that this is a problem – we can only update our "seconds" counter every two seconds.

But it's not a big problem: the Timer1 interrupt can simply increment the seconds count by two, and then the display routine needs to calculate which digit to display, by determining, at any point in time, whether more than 1 s has passed since the interrupt handler updated the seconds count. We can do that by testing the most significant bit of TMR1 (TMR1<15> = TMR1H<7>) – it changes every second, and if it's set to '1', it means that we're in an "odd" second, and the digit to display is one more than the current seconds count (which is always even).

This means that we'll need a variable to store the digit to be displayed, which we can define as:

```
unsigned char   digit;          // digit to display
```

We can then calculate the display digit:

```
        // calculate digit to display
        digit = t1_secs;          // get seconds count (even only)
                                  //   (maintained by Timer1 interrupt)
        if (TMR1H & 1<<7)         // if TMR1H<7> is set, we are in an odd second
            ++digit;              //   so add one to display digit
```

Or, if using HI-TECH C version 9.81, you may decide it is clearer to write:

```
        // calculate digit to display
        digit = t1_secs;          // get seconds count (even only)
                                  //   (maintained by Timer1 interrupt)
        if (TMR1 & 1<<15)         // if TMR1<15> is set, we are in an odd second
            ++digit;              //   so add one to display digit
```

We can then display this digit, using the `set7seg()` function developed in lesson 7:

```
        // display current seconds count
        set7seg(digit);             // output digit
```

This works with the lookup tables introduced in that lesson (modified for the different pin connections used in this example). They appear at the end of the program code, presented below.

You'll see that the `set7seg()` function in this example writes directly to PORTA and PORTC, instead of using shadow registers. This is ok because we're writing to the entire port, in a single operation; there is no opportunity for read-modify-write problems here.

All the ISR has to do is increment the seconds count by two, and handle the rollover from nine to zero:

```
    // increment seconds count by 2
    t1_secs += 2;
    if (t1_secs == 10)          // when count reaches 10,
        t1_secs = 0;            //   reset it to 0
```

Complete program

Here is the complete "seconds counter" program, for HI-TECH C v9.81:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 9, example 4                                 *
*                                                                       *
*   Demonstrates use of Timer1 LP oscillator and interrupt              *
*   to implement a single-digit seconds counter,                        *
*   shown on a 7-segment LED display                                    *
*                                                                       *
*   Single digit 7-segment LED display counts repeating 0 -> 9          *
*   1 count per second, with timing derived from 32.768 kHz crystal     *
*   driven by Timer1 oscillator                                         *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RA0-2, RC2-5 - 7-segment display bus (common cathode)           *
*       OSC1, OSC2   - 32.768 kHz crystal                               *
*                                                                       *
************************************************************************/

#include <htc.h>


/***** CONFIGURATION *****/
//  ext reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF & PWRTE_ON &
         FOSC_INTOSCIO & FCMEN_OFF & IESO_OFF);


/***** PROTOTYPES *****/
void set7seg(char digit);           // display digit on 7-segment display


/***** GLOBAL VARIABLES *****/
```

```
volatile unsigned char  t1_secs = 0;    // seconds count (even only),
                                        //  updated by Timer1 interrupt


/***** MAIN PROGRAM *****/
void main()
{
    unsigned char   digit;          // digit to display

    // Initialisation

    // configure ports
    PORTA = 0;                      // start with PORTA and PORTC clear
    PORTC = 0;                      //   (all LED segments off)
    TRISA = 0;                      // configure PORTA and PORTC as all outputs
    TRISC = 0;

    // configure timers
    T1CON = 0b00001010;             // configure Timer1:
            //-0------                   gate disabled (TMR1GE = 0)
            //--00----                   prescale = 1 (T1CKPS = 00)
            //----1---                   LP oscillator enabled (T1OSCEN = 1)
            //------1-                   external clock (TMR1CS = 1)
            //-------0                   disable Timer1 (TMR1ON = 0)
                                    //  -> increment TMR1 at 32.768 kHz
    TMR1 = 0;                       // clear TMR1 (start counting from 0)
    TMR1ON = 1;                     // start Timer1
    TMR1IE = 1;                     // enable Timer1 interrupt

    // enable interrupts
    PEIE = 1;                       // enable peripheral
    ei();                           //   and global interrupts


    // Main loop
    for (;;)
    {
        // calculate digit to display
        digit = t1_secs;          // get seconds count (even only)
                                  //   (maintained by Timer1 interrupt)
        if (TMR1 & 1<<15)         // if TMR1<15> is set, we are in an odd second
            ++digit;              //   so add one to display digit

        // display current seconds count
        set7seg(digit);             // output digit
    }
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    // *** Service Timer1 interrupt
    //  TMR1 overflows every 2 s
    //
    //  Handles background time keeping:
    //  Each interrupt advances current count by 2 secs
    //
    //  (only Timer1 interrupts are enabled)
    //
    TMR1IF = 0;                     // clear interrupt flag
```

```
    // increment seconds count by 2
    t1_secs += 2;
    if (t1_secs == 10)          // when count reaches 10,
        t1_secs = 0;            //   reset it to 0
}


/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[10] = {
        // RA2:0 = EFG
        0b000110,   // 0
        0b000000,   // 1
        0b000101,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b000111,   // 6
        0b000000,   // 7
        0b000111,   // 8
        0b000011    // 9
    };

    // Lookup pattern table for 7 segment display on PORTC
    const char pat7segC[10] = {
        // RC5:2 = ABCD
        0b111100,   // 0
        0b011000,   // 1
        0b110100,   // 2
        0b111100,   // 3
        0b011000,   // 4
        0b101100,   // 5
        0b101100,   // 6
        0b111000,   // 7
        0b111100,   // 8
        0b111100    // 9
    };

    // lookup pattern bits and write to port registers
    PORTA = pat7segA[digit];
    PORTC = pat7segC[digit];
}
```

Timer1 Operation in Sleep Mode

One of the more useful features of Timer1 is its ability to continue to operate when the PIC is in sleep mode.

Of course, this is only possible when Timer1 is externally driven by a signal on T1CKI or the LP crystal oscillator, because, when the PIC is in sleep mode, its instruction clock isn't running.

To operate Timer1 in sleep mode, it must be configured as an *asynchronous* counter ($\overline{\text{T1SYNC}}$ = 1), because it's not possible to synchronise Timer1 with the instruction clock, if the instruction clock isn't running.

You may want to operate Timer1 in sleep mode, to measure the time until external event wakes the device from sleep, especially if these events are far apart and you wish to conserve power while waiting for them.

Or, as is done in the Gooligum travel clock kit, you may wish to continue to keep track of the time while the circuit is in a low-power "standby" mode.

It is also possible to use Timer1 to periodically wake the PIC from sleep mode, with much greater precision than either the watchdog timer (see lesson 4) or the ultra low-power wake-up facility (described in midrange assembler lesson 10), when driven by an external clock or the LP oscillator.

For Timer1 to wake the PIC from sleep, the Timer1 interrupt must be enabled. When TMR1 overflows, the device will wake and, if global interrupts are also enabled (GIE = 1), an interrupt will be triggered.

With Timer1 driven by a 32.768 kHz oscillator, the PIC can wake at periods from 2 s (with no prescaler) to 16 s (with a 1:8 prescaler).

Example 5: Operating Timer1 in Sleep Mode

To demonstrate how to use Timer1 in sleep mode, we will modify the previous example, as follows:

- When the pushbutton is pressed, the PIC enters sleep mode and the display is blanked
- Timer1 continues to operate, updating the seconds count, while the PIC is in sleep mode
- When the button is pressed again, the PIC wakes up, and the updated count is displayed.

Firstly, Timer1 has to be configured as an asynchronous counter:

```
T1CON = 0b00001110;             // configure Timer1:
        //-0------                   gate disabled (TMR1GE = 0)
        //--00----                   prescale = 1 (T1CKPS = 00)
        //----1---                   LP oscillator enabled (T1OSCEN = 1)
        //-----1--                   asynchronous mode (/T1SYNC = 1)
        //------1-                   external clock (TMR1CS = 1)
        //-------0                   disable Timer1 (TMR1ON = 0)
                                //  -> increment TMR1 at 32.768 kHz
```

We also need to configure RA3 as an interrupt-on-change input, so that it can be used to wake the device from sleep (see lesson 4):

```
IOCA = 1<<nBUTTON;              // enable interrupt-on-change on pushbutton
RAIE = 1;                       // enable wake-up on PORTA change
```

With RAIE set, the device will wake from sleep if RA3 changes.

But RAIE is more than a "wake-up" enable; it is also an interrupt enable bit. To use Timer1 as an interrupt source we have to enable global interrupts – but this means that **all** enabled interrupt sources will trigger an interrupt, including changes on RA3.

This means that our interrupt service routine can no longer assume that Timer1 is the only interrupt source; we will need to differentiate between them, by checking the interrupt flags:

```
void interrupt isr(void)
{
    // Service all triggered interrupt sources

    if (TMR1IF)
    {
        // *** Service Timer1 interrupt
        // (Timer1 ISR goes here)
    }
```

```
    if (RAIF)
    {
        // *** Service PORTA change interrupt
        // (port change ISR goes here)
    }
}
```

We will also need a handler for PORTA change interrupts. It has nothing to do, because we don't really want to use PORTA change interrupts. But we do have to clear the interrupt flag (so that the interrupt doesn't immediately reoccur) and, importantly, clear the port-mismatch condition by reading PORTA, so that the interrupt flag isn't immediately set again:

```
        // *** Service PORTA change interrupt
        //
        //   PORTA interrupt-on-change is enabled
        //   only to allow wake-up from sleep on a button press,
        //   so do nothing in the interrupt handler
        //
        PORTA;                      // read PORTA to clear mismatch condition
        RAIF = 0;                   // clear interrupt flag
```

In the main loop, after displaying the current seconds count as before, we can test the pushbutton, and enter sleep mode if it is pressed:

```
        // test for enter standby mode
        if (!BUTTON)                // if button pressed (low)
        {
            PORTA = 0;              //   turn off display
            PORTC = 0;
            DbnceHi(BUTTON);        //   wait for stable button release
            RAIF = 0;               //   clear PORTA change interrupt flag

            SLEEP();                // sleep until button press
        }
```

The '`DbnceHi()`' macro from lesson 2 is used to ensure that the pushbutton has been released and is stable before the PIC enters sleep mode, to avoid a switch bounce causing an inadvertent wake-up.

Note that the PORTA change interrupt flag (RAIF) is explicitly cleared before entering sleep mode. Strictly speaking, this isn't necessary in this program because, every time RA3 changes (a pushbutton press, release, or a switch bounce), an interrupt is triggered, and the interrupt handler clears the port mismatch and clears RAIF. But, explicitly clearing RAIF in the main loop doesn't hurt, either.

When program execution continues, the device must have woken from sleep, which can only happen if TMR1 overflowed, triggering a Timer1 interrupt, or the pushbutton was pressed, triggering a PORTA change interrupt. We need to decide which type of interrupt it was, and go back to sleep if it was a Timer1 interrupt, or stay awake and continue the main loop, if the pushbutton was pressed.

We can't check the interrupt flags, because the ISR, which will have run immediately after the device woke from sleep, will have cleared whichever flag was active. But there's an easy solution – if the pushbutton was pressed, it will still be down for some time after the device wakes, because PICs can execute instructions much faster than humans can release pushbuttons. So we can simply check whether the pushbutton is still down and go back to sleep if so, by using a `do … while` construct, as follows:

```
        do                          //   sleep until button press
            SLEEP();
        while (BUTTON);
```

The sleep / wake-up code then becomes:

```
        // test for enter standby mode
        if (!BUTTON)                // if button pressed (low)
        {
            PORTA = 0;              //   turn off display
            PORTC = 0;
            DbnceHi(BUTTON);        //   wait for stable button release
            RAIF = 0;               //   clear PORTA change interrupt flag
            do                      //   sleep until button press
            {
                SLEEP();
                // if we get here, we've been woken by Timer1 or button press,
                // so go back to sleep unless the button is pressed
            }
            while (BUTTON);
            // button pressed, so stay awake
            DbnceHi(BUTTON);        //   wait for stable button high (= up)
        }
```

Note that the pushbutton is debounced, following wake-up, to ensure that the device doesn't prematurely re-enter sleep mode (the next time around the main loop) due to a contact bounce.

In a larger application, you wouldn't handle the button press detection and debouncing in this way, because the debounce routine blocks the main loop – if you hold down the pushbutton, the display will remain blank until you release it, which is not what you'd usually want. Scanning and debouncing switches would usually be handled by a timer interrupt, perhaps Timer0, running every 1 ms or so, as we saw in lesson 3.

Complete program

Although this example was based on the previous (seconds counter) example code, there have been enough changes throughout the code that it's worth seeing the complete listing, so see how the changes fit in.

Here is the PICC-Lite version:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 9, example 5                                 *
*                                                                       *
*   Demonstrates use of Timer1 LP oscillator during sleep mode          *
*                                                                       *
*   Single digit 7-segment LED display counts repeating 0 -> 9          *
*   1 count per second, with timing derived from 32.768 kHz crystal     *
*   driven by Timer1 oscillator                                         *
*                                                                       *
*   Device enters sleep mode (and display is blanked) when pushbutton   *
*   is pressed.                                                         *
*   Timer continues to count in sleep mode, maintaining seconds count.  *
*   Device wakes from sleep mode when button is pressed again,          *
*   resuming display and seconds count from the current value.          *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RA0-2, RC2-5 - 7-segment display bus (common cathode)           *
*       RA3          - sleep/wake pushbutton (active low)               *
*       OSC1, OSC2   - 32.768 kHz crystal                               *
*                                                                       *
************************************************************************/
```

```
#include <htc.h>

#include "dbmacros-PCL.h"   // DbnceHi() - debounce switch, wait for high
                            //           - uses TMR0 at 256us/tick


/***** CONFIGURATION *****/
//  int reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLRDIS & UNPROTECT & BORDIS & WDTDIS & PWRTEN & INTIO & FCMDIS &
         IESODIS);

// Pin assignments
#define BUTTON  RA3         // sleep/wake pushbutton (active low)
#define nBUTTON 3           //   (pin 3)


/***** PROTOTYPES *****/
void set7seg(char digit);           // display digit on 7-segment display


/***** GLOBAL VARIABLES *****/
volatile unsigned char  t1_secs = 0;    // seconds count (even only),
                                        //  updated by Timer1 interrupt


/***** MAIN PROGRAM *****/
void main()
{
    unsigned char   digit;          // digit to display

    // Initialisation

    // configure ports
    PORTA = 0;                      // start with PORTA and PORTC clear
    PORTC = 0;                      //   (all LED segments off)
    TRISA = 0;                      // configure PORTA and PORTC as all outputs
    TRISC = 0;
    IOCA = 1<<nBUTTON;              // enable interrupt-on-change on pushbutton
    RAIE = 1;                       // enable wake-up on PORTA change

    // configure Timer0 (for DbnceHi() macro)
    OPTION = 0b11010111;            // configure Timer0:
             //--0-----                  timer mode (T0CS = 0)
             //----0---                  prescaler assigned to Timer0 (PSA = 0)
             //-----111                  prescale = 256 (PS = 111)
                                    //  -> increment TMR0 every 256 us

    // configure Timer1
    T1CON = 0b00001110;             // configure Timer1:
            //-0------                   gate disabled (TMR1GE = 0)
            //--00----                   prescale = 1 (T1CKPS = 00)
            //----1---                   LP oscillator enabled (T1OSCEN = 1)
            //-----1--                   asynchronous mode (/T1SYNC = 1)
            //------1-                   external clock (TMR1CS = 1)
            //-------0                   disable Timer1 (TMR1ON = 0)
                                    //  -> increment TMR1 at 32.768 kHz
    TMR1H = 0;                      // clear TMR1 (start counting from 0)
    TMR1L = 0;
    TMR1ON = 1;                     // start Timer1
```

```
    TMR1IE = 1;                     // enable Timer1 interrupt

    // enable interrupts
    PEIE = 1;                       // enable peripheral
    ei();                           //   and global interrupts


    // Main loop
    for (;;)
    {
        // calculate digit to display
        digit = t1_secs;          // get seconds count (even only)
                                  //   (maintained by Timer1 interrupt)
        if (TMR1H & 1<<7)         // if TMR1H<7> is set, we are in an odd second
            ++digit;              //   so add one to display digit

        // display current seconds count
        set7seg(digit);           // output digit

        // test for enter standby mode
        if (!BUTTON)              // if button pressed (low)
        {
            PORTA = 0;            //   turn off display
            PORTC = 0;
            DbnceHi(BUTTON);      //   wait for stable button release
            RAIF = 0;             //   clear PORTA change interrupt flag
            do                    //   sleep until button press
            {
                SLEEP();
                // if we get here, we've been woken by Timer1 or button press,
                // so go back to sleep unless the button is pressed
            }
            while (BUTTON);
            // button pressed, so stay awake
            DbnceHi(BUTTON);      //   wait for stable button high (= up)
        }
    }
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    // Service all triggered interrupt sources

    if (TMR1IF)
    {
        // *** Service Timer1 interrupt
        //   TMR1 overflows every 2 s
        //
        //   Handles background time keeping:
        //   Each interrupt advances current count by 2 secs
        //
        TMR1IF = 0;               // clear interrupt flag

        // increment seconds count by 2
        t1_secs += 2;
        if (t1_secs == 10)        // when count reaches 10,
            t1_secs = 0;          //   reset it to 0
    }
```

```
    if (RAIF)
    {
        // *** Service PORTA change interrupt
        //
        //   PORTA interrupt-on-change is enabled
        //   only to allow wake-up from sleep on a button press,
        //   so do nothing in the interrupt handler
        //
        PORTA;                      // read PORTA to clear mismatch condition
        RAIF = 0;                  // clear interrupt flag
    }
}


/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[10] = {
        // RA2:0 = EFG
        0b000110,   // 0
        0b000000,   // 1
        0b000101,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b000111,   // 6
        0b000000,   // 7
        0b000111,   // 8
        0b000011    // 9
    };

    // Lookup pattern table for 7 segment display on PORTC
    const char pat7segC[10] = {
        // RC5:2 = ABCD
        0b111100,   // 0
        0b011000,   // 1
        0b110100,   // 2
        0b111100,   // 3
        0b011000,   // 4
        0b101100,   // 5
        0b101100,   // 6
        0b111000,   // 7
        0b111100,   // 8
        0b111100    // 9
    };

    // lookup pattern bits and write to port registers
    PORTA = pat7segA[digit];
    PORTC = pat7segC[digit];
}
```

Timer1 Gate Control

One of the key features that differentiate Timer1 from Timer0 is *gate control*[6].

In this mode, the timer is controlled by either an external digital signal, or an analog input connected to comparator 2. The timer will only increment when an appropriate signal is present.

This makes it possible to accurately time external events, and measure pulse widths, with a resolution limited only by the timer clock period, which, with an external clock source, can be as little as 60 ns[7].

To enable gate control, set the TMR1GE bit in the T1CON register.

By default, when gate control is enabled (TMR1GE = 1), Timer1 will only increment when the $\overline{\text{T1G}}$ digital input (which is on the same pin as RA4 on the 16F684) is held low.

The output of comparator 2 (C2OUT) can also be used to gate Timer1, allowing the timer to be controlled by an analog input. Timer1 will then increment only when the analog signal is above or below a threshold, depending on how comparator 2 is configured (see lesson 6).

To select comparator mode, clear the T1GSS bit in the CMCON1 register.

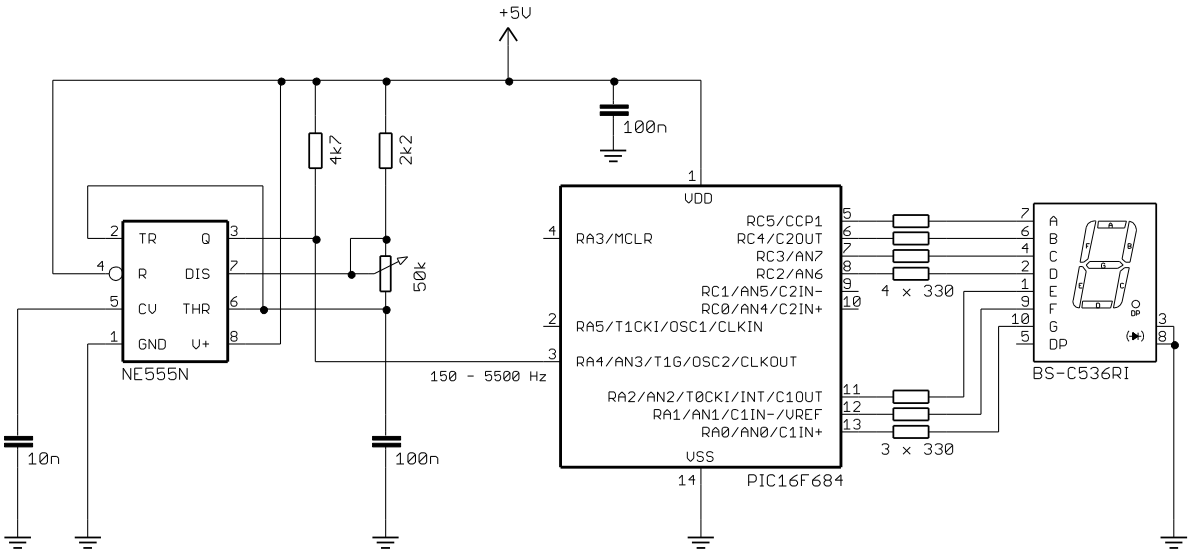
Whenever comparator gating is selected, the C2SYNC bit should be set, to synchronise the comparator output with the Timer1 clock. This ensures that no timer increments will be lost, if the comparator changes during an increment.

To measure active-high time, by enabling Timer1 when the gate signal (either $\overline{\text{T1G}}$ or C2OUT, as selected by the T1GSS bit) is high instead of low, set the T1GINV Timer1 gate inversion bit in the T1CON register.

A couple of examples will help to illustrate this.

Example 6: Pulse width measurement using Timer1 gate control

To show how Timer1 gate control can be used to measure the pulse width of a digital signal, we'll use the circuit shown below:



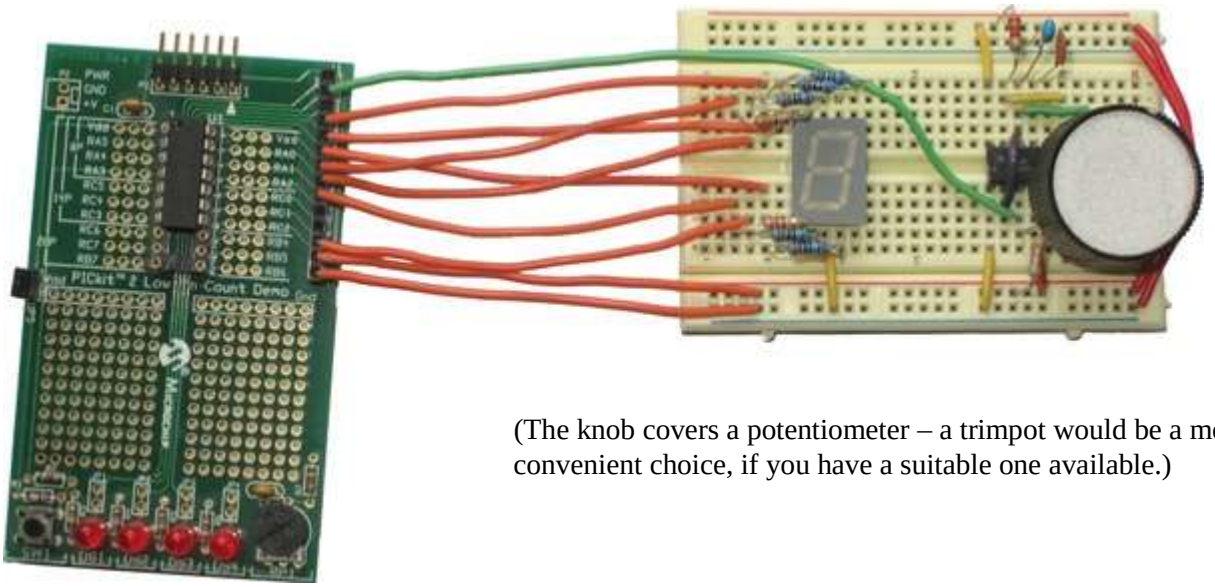
[6] The Timer1 module in some older midrange PICs does not provide this feature – but all modern devices do.
[7] for most midrange PICs including the 16F684, in asynchronous counter mode

The 555 timer generates a train of digital pulses, with a frequency between approximately 150 and 5500 Hz, with (active-high) pulses ranging from approx 170 µs to 3300 µs.

The measured pulse width is displayed as single hexadecimal digit on the 7-segment LED display – after appropriate scaling, of course!

Given that this is only an example, we can simplify the problem by displaying the pulse width in arbitrary units. And the easiest unit to use is 256 µs, because dividing by 256 on an 8-bit device is very simple: just ignore the least significant byte

This circuit can be built using prototyping breadboard connected to the 14-pin header on the LPC Demo Board, as pictured:



(The knob covers a potentiometer – a trimpot would be a more convenient choice, if you have a suitable one available.)

The code is quite straightforward.

First, we must configure the $\overline{\text{T1G}}$ pin (shared with RA4) as a digital input[8]:

```
// configure ports
PORTA = 0;                      // start with PORTA and PORTC clear
PORTC = 0;                      //   (all LED segments off)
TRISC = 0;                      // configure PORTA and PORTC as all outputs
TRISA = 1<<nT1G;                //   except T1G/RA4 input
ANSEL = 0;                      // no analog inputs
```

where we have defined:

```
// Pin assignments
#define T1G     RA4         // Timer1 gate input (configured as active high)
#define nT1G    4           //   (pin 4)
```

[8] If RA4 is acting as an output, it will interfere with the gate control signal. If RA4 is configured as an analog input, $\overline{\text{T1G}}$ will always read as '0'. So, RA4 must be configured as a digital input.

Next, we configure Timer1 with gate control enabled for an active-high signal on $\overline{\text{T1G}}$:

```
// configure timers
T1CON = 0b11000001;             // configure Timer1:
        //1-------                   gate is active high (T1GINV = 1)
        //-1------                   gate enabled (TMR1GE = 1)
        //--00----                   prescale = 1 (T1CKPS = 00)
        //----0---                   LP oscillator disabled (T1OSCEN = 0)
        //------0-                   internal clock (TMR1CS = 0)
        //-------1                   enable Timer1 (TMR1ON = 1)
                                  //  -> increment TMR1 every 1 us,
T1GSS = 1;                        //     gated by active high on /T1G
```

Note that, since digital input gating is the default, it is not really necessary to explicitly set T1GSS, as is done here. But, for the sake of clarity and maintainability, it doesn't hurt…

To measure each pulse width, we first make the assumption that the gate signal is not present ($\overline{\text{T1G}}$ is low). The first time the program runs, this may not be true – but as you'll see, by the time the main loop first completes, we be sure that $\overline{\text{T1G}}$ is low. Although the display may be incorrect when the program starts, it will only be incorrect for a few milliseconds at most, which is ok in this application.

With no gate signal present, the timer cannot be counting, so we can safely clear it:

```
TMR1L = 0;                  // zero TMR1
TMR1H = 0;
```

Or, if using HI-TECH C v9.81, you can write:

```
TMR1 = 0;
```

We can then wait for the signal on $\overline{\text{T1G}}$ to go high, and then low again:

```
// wait for one full high pulse
while (!T1G)                // wait for T1G to go high
    ;
while (T1G)                 // wait for T1G to go low
    ;
```

During this time, Timer1 will have been counting only while the input signal was high. Since, at this point, the signal is now low again (the pulse has passed), we know that Timer1 has stopped counting, and holds a value equal to the width of the pulse, to the nearest microsecond (because it was configured to increment every 1 µs).

Of course, this is only true if the timer didn't overflow while the gate input was high. If you're measuring pulses more than 65 ms wide, and you need microsecond resolution, you'll need to provide for timer overflows; a timer interrupt would be an easy way to do this. But in this example, our pulses are no more than 4 ms wide, so we don't have that problem.

Now that we have the pulse width in TMR1, we can retrieve it and display a scaled value.

Since we are scaling the result by dividing it by 256, we can simply ignore the least significant byte, which is TMR1L in this case.

The scaled result, with a range of 0 to 15, is then found in the low nybble of TMR1H, and we can extract it and display it as a single hexadecimal digit on the 7-segment display, using our standard '`set7seg()`' function:

```
// display scaled Timer1 count (divide by 256)
set7seg(TMR1H & 0x0f);      // display low nybble of TMR1H
```

Or, if you are using HI-TECH C v9.81, you may choose to express this as:

```
set7seg(TMR1>>8 & 0x0f);    // display low nybble of TMR1/256
```

or even:

```
set7seg(TMR1/256 & 0x0f);    // display low nybble of TMR1/256
```

to make it clear that we're displaying TMR1/256.

Note that each of these expressions will only work if the pulse width is less than 4096 µs, which, using the component values shown in the circuit above, will always be true. In a real application, you would check to see whether limits such as this had been exceeded.

At this point, less than 50 µs will have passed since we detected the end of the active-high pulse, so, given the relatively low-speed pulses used in this example, it is safe to assume that the input signal is still low, and we can restart the loop, resetting the timer, ready to measure the next pulse.

Complete program

Here is how these code fragments fit together, using HI-TECH C v9.81:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 9, example 6                                 *
*                                                                       *
*   Demonstrates use of Timer1 gate control                             *
*   to measure the pulse width of a digital signal on /T1G,             *
*   scaled and displayed as a single hex digit                          *
*                                                                       *
*   Timer1 is used to time (in microseconds) every high pulse on /T1G.  *
*   Result is divided by 256 and displayed in hex on a single-digit     *
*   7-segment LED display.                                              *
*   Time base is internal RC oscillator.                                *
*                                                                       *
************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       RA0-2, RC2-5 - 7-segment display bus (common cathode)           *
*       /T1G (=RA4)  - signal to measure pulse width of                 *
*                      (active high, 4 ms max)                          *
*                                                                       *
************************************************************************/

#include <htc.h>

/***** CONFIGURATION *****/
//  ext reset, no code or data protect, no brownout detect,
//  no watchdog, power-up timer, int clock with I/O,
//  no failsafe clock monitor, two-speed start-up disabled
__CONFIG(MCLRE_ON & CP_OFF & CPD_OFF & BOREN_OFF & WDTE_OFF & PWRTE_ON &
         FOSC_INTOSCIO & FCMEN_OFF & IESO_OFF);
```

```
// Pin assignments
#define T1G     RA4         // Timer1 gate input (configured as active high)
#define nT1G    4           //   (pin 4)


/***** PROTOTYPES *****/
void set7seg(char digit);           // display digit on 7-segment display


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

    // configure ports
    PORTA = 0;                      // start with PORTA and PORTC clear
    PORTC = 0;                      //   (all LED segments off)
    TRISC = 0;                      // configure PORTA and PORTC as all outputs
    TRISA = 1<<nT1G;                //   except T1G/RA4 input
    ANSEL = 0;                      // no analog inputs

    // configure timers
    T1CON = 0b11000001;             // configure Timer1:
            //1-------                   gate is active high (T1GINV = 1)
            //-1------                   gate enabled (TMR1GE = 1)
            //--00----                   prescale = 1 (T1CKPS = 00)
            //----0---                   LP oscillator disabled (T1OSCEN = 0)
            //------0-                   internal clock (TMR1CS = 0)
            //-------1                   enable Timer1 (TMR1ON = 1)
                                    //  -> increment TMR1 every 1 us,
    T1GSS = 1;                      //     gated by active high on /T1G


    // Main loop
    for (;;)
    {
        // Measure width of pulses on /T1G input
        // (assume /T1G is low at start of loop)

        // clear Timer1
        TMR1 = 0;

        // wait for one full high pulse
        while (!T1G)                // wait for T1G to go high
            ;
        while (T1G)                 // wait for T1G to go low
            ;

        // display scaled Timer1 count (divide by 256)
        set7seg(TMR1/256 & 0x0f);    // display low nybble of TMR1/256
    }
}


/***** FUNCTIONS *****/

/***** Display digit on 7-segment display *****/
void set7seg(char digit)
{
    // Lookup pattern table for 7 segment display on PORTA
    const char pat7segA[16] = {
```

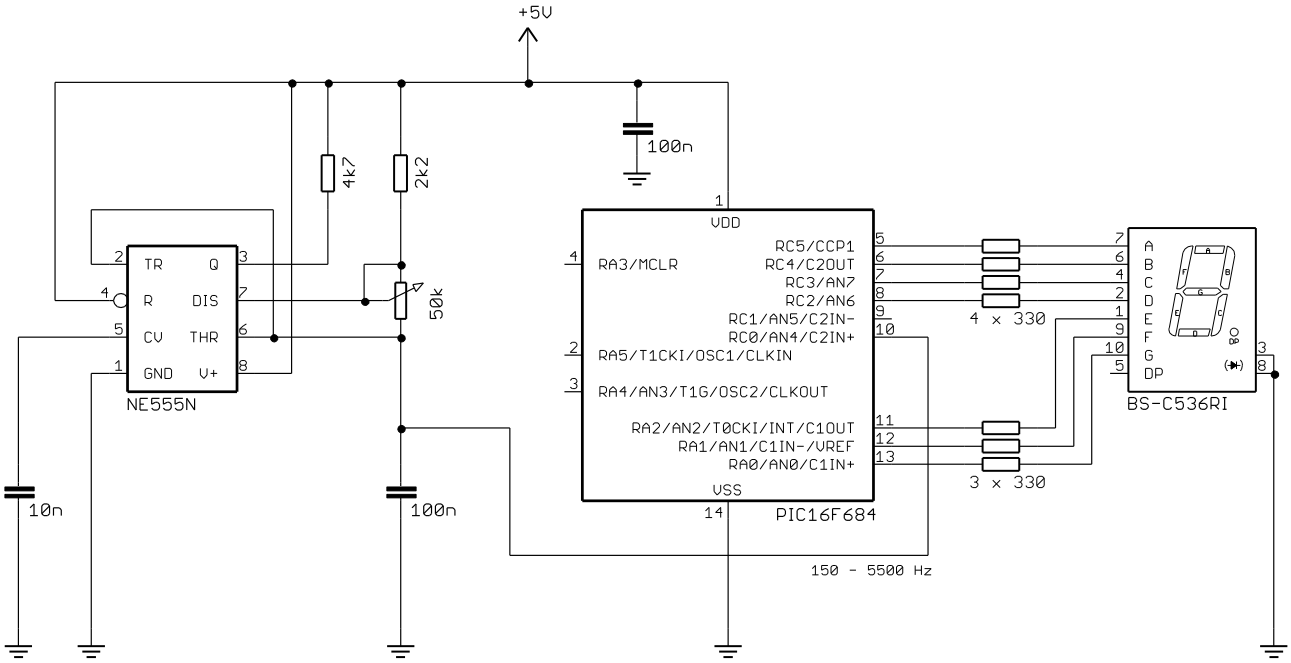
```
        // RA2:0 = EFG
        0b000110,   // 0
        0b000000,   // 1
        0b000101,   // 2
        0b000001,   // 3
        0b000011,   // 4
        0b000011,   // 5
        0b000111,   // 6
        0b000000,   // 7
        0b000111,   // 8
        0b000011,   // 9
        0b000111,   // A
        0b000111,   // b
        0b000110,   // C
        0b000101,   // d
        0b000111,   // E
        0b000111    // F
    };

    // Lookup pattern table for 7 segment display on PORTC
    const char pat7segC[16] = {
        // RC5:2 = ABCD
        0b111100,   // 0
        0b011000,   // 1
        0b110100,   // 2
        0b111100,   // 3
        0b011000,   // 4
        0b101100,   // 5
        0b101100,   // 6
        0b111000,   // 7
        0b111100,   // 8
        0b111100,   // 9
        0b111000,   // A
        0b001100,   // b
        0b100100,   // C
        0b011100,   // d
        0b100100,   // E
        0b100000    // F
    };

    // lookup pattern bits and write to port registers
    PORTA = pat7segA[digit];
    PORTC = pat7segC[digit];
}
```

Example 7: Pulse width measurement using Timer1 comparator gate control

It is also possible to use an analog signal to gate Timer1. To illustrate this, we'll re-implement the previous example, this time measuring the pulse width of an analog signal via a comparator input.

The circuit is essentially the same as before, except that the input signal is now taken from pin 6 of the 555 timer, which has a sawtooth waveform ramping smoothly between ⅓ and ⅔ of V_{CC} (5 V here).

This signal is connected to the C2IN+ comparator 2 input, as shown on the next page:



The code is very similar to that in the previous example; the only differences being that we need to configure the comparator, select the comparator gate control mode, and poll the comparator output instead of the digital input.

First, we need to ensure that C2IN+, which shares its pin with RC0, is not configured as a digital output. Note that we do not need to configure it as an analog input, because we are using the comparator, not the ADC. So, to configure the ports, we have:

```
PORTA = 0;                      // start with PORTA and PORTC clear
PORTC = 0;                      //   (all LED segments off)
TRISA = 0;                      // configure PORTA and PORTC as all outputs
TRISC = 0b000001;               //   except C2IN+ input (RC0)
```

When measuring the width of an analog signal, we need a reference level, or threshold, to compare the signal to; the "pulse width" is then the period during which the signal remains higher than the reference.

Since our analog input signal is varying between ⅓ and ⅔ of the supply voltage, it makes sense to use a reference level of ½ of V_{DD}.

We can use the programmable voltage reference to generate this threshold:

```
VRCON = 0b10101100;             // configure voltage reference:
        //1-------                  voltage ref enabled (VREN = 1)
        //--1-----                  CVref = 0.5 x Vdd   (VRR = 1,
        //----1100                                       VR = 12)
                                //  -> CVref = 2.5 V, if Vdd = 5.0 V
```

Comparator 2 is then configured so that its output (C2OUT) goes high whenever C2IN+ is higher than the reference:

```
CMCON0 = 0b00101010;            // configure comparators:
         //--1-----                 C2 output inverted (C2INV = 1)
         //----1---                 C2 -ref is C2IN+   (CIS = 1,
         //-----010                 C2 +ref is CVref     CM = 010)
                                //  -> C2OUT = 1 if C2IN+ > CVref
```

Since we're using C2OUT as the gate source for Timer1, we need to synchronise comparator 2 with the Timer1 clock:

```
    C2SYNC = 1;                     //  sync C2OUT with TMR1
```

Timer1 is configured exactly as before, except that we need to clear T1GSS to select comparator gating:

```
    T1CON = 0b11000001;             // configure Timer1:
            //1-------                  gate is active high (T1GINV = 1)
            //-1------                  gate enabled (TMR1GE = 1)
            //--00----                  prescale = 1 (T1CKPS = 00)
            //----0---                  LP oscillator disabled (T1OSCEN = 0)
            //------0-                  internal clock (TMR1CS = 0)
            //-------1                  enable Timer1 (TMR1ON = 1)
                                    //  -> increment TMR1 every 1 us,
    T1GSS = 0;                      //     gated by C2OUT (active high)
```

TMR1 will now increment only when C2OUT is high, i.e. when the analog input is higher than 2.5 V.

The main loop is the same as in the previous example, except that we need to poll C2OUT instead of $\overline{\text{T1G}}$:

```
        // wait for one full high pulse
        while (!C2OUT)              // wait for C2OUT to go high
            ;
        while (C2OUT)               // wait for C2OUT to go low
            ;
```

Main program (initialisation and main loop)

Other than the changes and additions described above, the program is nearly identical to the previous example. Nevertheless, here is the main program code, for HI-TECH C v 9.81, showing where these changes fit in:

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

    // configure ports
    PORTA = 0;                      // start with PORTA and PORTC clear
    PORTC = 0;                      //   (all LED segments off)
    TRISA = 0;                      // configure PORTA and PORTC as all outputs
    TRISC = 0b000001;               //   except C2IN+ input (RC0)

    // configure voltage reference
    VRCON = 0b10101100;             // configure voltage reference:
            //1-------                  voltage ref enabled (VREN = 1)
            //--1-----                  CVref = 0.5 x Vdd   (VRR = 1,
            //----1100                                       VR = 12)
                                    //  -> CVref = 2.5 V, if Vdd = 5.0 V

    // configure comparators
    CMCON0 = 0b00101010;            // configure comparators:
             //--1-----                  C2 output inverted (C2INV = 1)
             //----1---                  C2 -ref is C2IN+   (CIS = 1,
             //-----010                  C2 +ref is CVref     CM = 010)
                                    //  -> C2OUT = 1 if C2IN+ > CVref
    C2SYNC = 1;                     //  sync C2OUT with TMR1
```

```
    // configure timers
    T1CON = 0b11000001;             // configure Timer1:
            //1-------                  gate is active high (T1GINV = 1)
            //-1------                  gate enabled (TMR1GE = 1)
            //--00----                  prescale = 1 (T1CKPS = 00)
            //----0---                  LP oscillator disabled (T1OSCEN = 0)
            //------0-                  internal clock (TMR1CS = 0)
            //-------1                  enable Timer1 (TMR1ON = 1)
                                    //  -> increment TMR1 every 1 us,
    T1GSS = 0;                      //     gated by C2OUT (active high)


    // Main loop
    for (;;)
    {
        // Measure width of pulses on C2IN+ input
        // (assume C2 output is low at start of loop)

        // clear Timer1
        TMR1 = 0;

        // wait for one full high pulse
        while (!C2OUT)              // wait for C2OUT to go high
            ;
        while (C2OUT)               // wait for C2OUT to go low
            ;

        // display scaled Timer1 count (divide by 256)
        set7seg(TMR1/256 & 0x0f);    // display low nybble of TMR1/256
    }
}
```

Hopefully this lesson has illustrated the enhanced usefulness of Timer1 (compared with Timer0), especially its ability to operate in sleep mode and the availability of gate control – and demonstrated that it is easy to use HI-TECH C to access these features.

The midrange PIC architecture includes one more timer: the 8-bit Timer2. It's not as complex as Timer1, but provides some very useful functionality of its own, which we'll look at briefly in the next lesson.