Introduction to PIC Programming

Programming Midrange PICs in C

by David Meiklejohn, Gooligum Electronics

Lesson 6: Analog Comparators, part 1

Analog comparators are used to detect whether an input signal is above or below a threshold, which may be fixed (generated by a voltage reference) or variable (perhaps another analog input).

Midrange lessons 9 and 10 explained how to use the type of comparator module and programmable voltage reference available on older midrange PIC devices[1], such as the PIC12F629 and PIC16F684, using assembly language. This lesson demonstrates how to use C to access those facilities, re-implementing the examples from those assembler lessons using the free HI-TECH C[2] (in "Lite" mode) and PICC-Lite compilers, as usual.

In summary, this lesson covers:

- Basic use of the comparator modules available on the PIC12F629 and PIC16F684
- Adding comparator hysteresis
- Comparator-driven interrupts
- Wake-up on comparator change with single and dual comparators
- Using the programmable voltage reference
- Using a single comparator to:
    - Compare a signal against a range of input thresholds
    - Compare two independent inputs against a common reference
- Using a dual comparator module with four independent inputs

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

Comparators

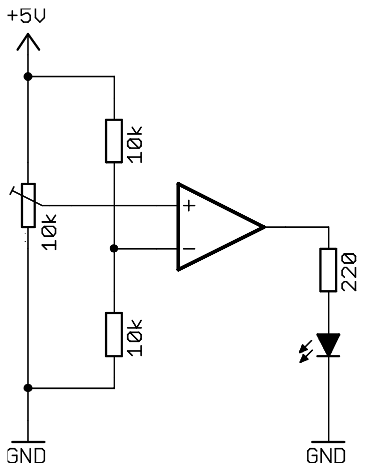
As we saw in midrange lesson 9, an *analog comparator* is a device which compares the voltages present on its positive and negative inputs. In normal (non-inverted) operation, the comparator's output is set to a logical "high" only when the voltage on the positive input is greater than that on the negative input; otherwise the output is "low". As such, they provide an interface between analog and digital circuitry.

[1] Newer midrange PICs, such as the 16F690 and 16F887, include a more flexible type of comparator module, which will be covered in a later lesson.

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

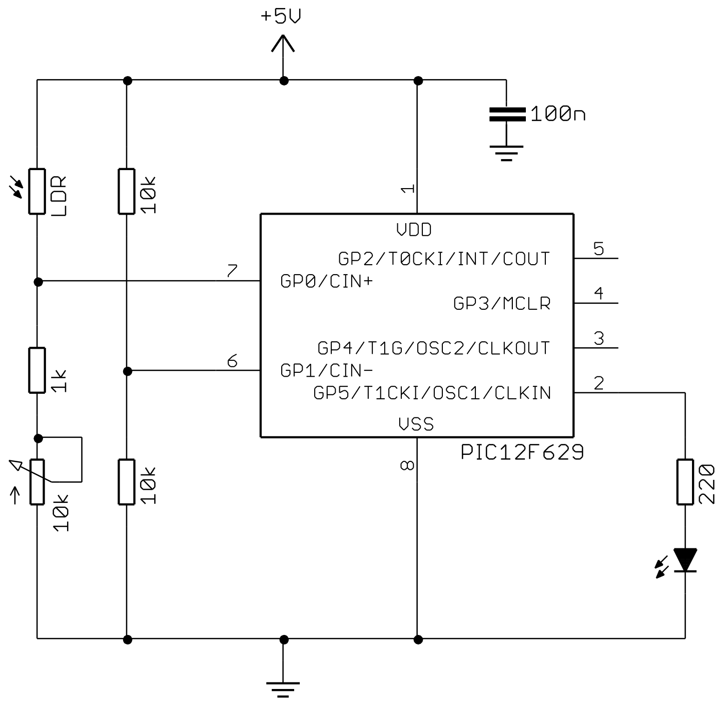
In the circuit shown on the right, the comparator output will go high, lighting the LED, only when the potentiometer is set to a position past "half-way", i.e. positive input is greater than 2.5 V.

Comparators are typically used to detect when an analog input is above or below some threshold (or, if two comparators are used, within a defined band) – very useful for working with many types of real-world sensors. They are also used with digital inputs to match different logic levels, and to shape poorly defined signals.



PIC12F629 Comparator Module

In midrange lesson 9, the circuit on the right, which includes a light dependent resistor (LDR), was used to demonstrate the basic operation of the PIC12F629's single comparator modules.



Note that if you do not have an LDR available, you can still explore comparator operation by connecting the CIN+ input directly to the centre tap on the 10 kΩ potentiometer.

If you are building this circuit using Microchip's Low Pin Count Demo Board, you will find that the centre tap on the 10 kΩ pot on the demo board is already connected to CIN+ via a 1 kΩ resistor. This doesn't appreciably affect the voltage from the pot, because the comparator inputs are very high impedance. Note that if you wish to reconfigure the circuit to include an LDR, as shown, you need to remove jumper JP5 from the LPC demo board (which will involve cutting the PCB trace across it, if you have not already modified your board), to disconnect the potentiometer from the +5 V supply[3]. Note as well that the CIN+ pin is also used for programming, and so is loaded by the PICkit 2 programmer if it is connected to the demo board, slightly affecting the operation of this circuit.

We saw in midrange lesson 9 that, to configure the 12F629's comparator to behave like a simple standalone comparator, so that the output bit (COUT) is high if the voltage on the CIN+ input is higher than that on the

[3] The +5 V supply connected through JP5 is also used to hold $\overline{\text{MCLR}}$ high when the reset line is tri-stated – as it will be if you are using a PICkit 2 with MPLAB and have selected the '3-State on "Release from Reset"' setting. For correct operation with JP5 removed, you must either disable this setting, or use internal $\overline{\text{MCLR}}$.

CIN- input, it is necessary to select comparator mode 2 by setting the CM bits in the CMCON register to '010' and clearing the CINV bit, so that the output is not inverted:

```
        movlw   0<<CINV|b'010'
                                ; select mode 2 (CM = 010):
                                ;   +ref is CIN+, -ref is CIN-,
                                ;   comparator on, no external output
                                ; output not inverted (CINV = 0)
        banksel CMCON           ;  -> COUT = 1 if CIN+ > CIN-
        movwf   CMCON
```

The LED attached to GP5 was turned on when the comparator output was high (COUT = 1) by using a shadow register, as follows:

```
loop    clrf    sGPIO           ; assume COUT = 0 -> LED off
        banksel CMCON
        btfsc   CMCON,COUT      ; if comparator output high
        bsf     sGPIO,nLED      ;   turn on LED

        movf    sGPIO,w         ; copy shadow to GPIO
        banksel GPIO
        movwf   GPIO

        goto    loop            ; repeat forever
```

HI-TECH C PRO or PICC-Lite

As usual, the include files provided with the HI-TECH C compilers define the bits comprising the CMCON register as single-bit variables:

```
/* CMCON Bits */
         bit      CM0   @ (unsigned)&CMCON*8+0;
         bit      CM1   @ (unsigned)&CMCON*8+1;
         bit      CM2   @ (unsigned)&CMCON*8+2;
         bit      CIS   @ (unsigned)&CMCON*8+3;
         bit      CINV  @ (unsigned)&CMCON*8+4;
volatile bit      COUT  @ (unsigned)&CMCON*8+6;
```

Unfortunately, the CM (comparator mode) bits cannot be accessed as a single 3-bit field, so to use these symbols to select mode 2, by setting CM<2:0> to 010, you must write:

```
    CM2 = 0; CM1 = 1; CM0 = 0;     // CM = 010 (comparator mode 2)
```

However, there is a trick you can use: since the CM field is the least significant three bits of the CMCON register, you can setup the CM bits by writing the mode value to CMCON:

```
    CMCON = 0b010;                 // select comparator mode 2 (CM = 010)
```

Or, you may wish to express this in decimal:

```
    CMCON = 2;                     // select comparator mode 2 (CM = 010)
```

Although it's your choice, and you will often see examples using the decimal form ('`CMCON = 7`' is commonly used to disable the comparator), the PIC data sheets specify the comparator modes in binary, so that is the format we will use here.

Either way, this has the side effect of clearing all the upper bits of CMCON, but that's ok if this is the first of a series of instructions initialising CMCON, with subsequent instructions setting or clearing the upper bits.

Of course, strictly speaking, there is no need to clear bits which have already been cleared, but it makes your code easier to follow if you make it explicit.

So to initialise the comparator, we have:

```
    // configure comparator
    CMCON = 0b010;             // select mode 2 (CM = 010):
                               //   +ref is CIN+, -ref is CIN-,
                               //   comparator on, no external output
    CINV = 0;                  // output not inverted
                               // -> COUT = 1 if CIN+ > CIN-
```

Note that the remaining bits in CMCON are not being explicitly set or cleared; that is ok because in this example we don't care what values they are set to.

Alternatively, you could write this as a single statement:

```
    CMCON = 0b00000010;       // configure comparator 1:
            //-----010            select mode 2 (CM = 010):
            //                      +ref is CIN+, -ref is CIN-,
            //                      comparator on, no external output
            //---0----            output not inverted (CINV = 0)
            //                    -> C1OUT = 1 if CIN+ > CIN-
```

This is ok, as long as you express the value in binary, so that it is obvious which bits are being set or cleared, and clearly commented, as above.

The first form is clear and easy to maintain, and is therefore the method usually adopted in these lessons. On the other hand, a series of assignments like this requires more program memory than a single whole-register assignment. It is also no longer an *atomic* operation, where all the bits are updated at once. This can be an important consideration in some instances, but it is not relevant here.

The comparator's output bit, COUT, is available as the single-bit variable '`COUT`'. For example:

```
        LED = COUT;            // turn on LED iff comparator output high
```

With the above configuration, the LED will turn on when the LDR is illuminated (you may need to adjust the pot to set the threshold appropriately for your ambient lighting).

If instead you wanted it to operate the other way, so that the LED is lit when the LDR is in darkness, you could invert the comparator output test, so that the LED is set high when COUT is low:

```
        LED = !COUT;           // turn on LED iff comparator output low
```

Alternatively, you can configure the comparator so that its output is inverted, using

```
    // configure comparator
    CMCON = 0b010;             // select mode 2 (CM = 010):
                               //   +ref is CIN+, -ref is CIN-,
                               //   comparator on, no external output
    CINV = 1;                  // inverted output
                               // -> COUT = 1 if CIN+ < CIN-
```

Complete program

Here is the complete inverted output version of the program, for HI-TECH C:

```
/************************************************************************
*                                                                       *
*   Description:    Lesson 6, example 1b                                *
*                                                                       *
*   Demonstrates use of comparator output inversion bit                 *
*                                                                       *
*   Turns on LED when voltage on CIN+ < voltage on CIN-                 *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       CIN+  - voltage to be measured (e.g. pot output or LDR)         *
*       CIN-  - threshold voltage (set by voltage divider resistors)    *
*       GP5   - indicator LED                                           *
*                                                                       *
************************************************************************/

#include <htc.h>


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

// Pin assignments
#define LED     GPIO5           // indicator LED on GP5
#define nLED    5               //   (port bit 3)


/***** MAIN PROGRAM *****/
void main()
{
    // Initialisation
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as an output

    // configure comparator
    CMCON = 0b010;              // select mode 2 (CM = 010):
                                //   +ref is CIN+, -ref is CIN-,
                                //   comparator on, no external output
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if CIN+ < CIN-

    // Main loop
    for (;;)
    {
        LED = COUT;             // continually display comparator output
    }
}
```

Comparisons

The table on the next page summarises the resource usage for the “comparator inverted output” assembler and C examples, along with the baseline (PIC16F506) versions of this example, from baseline C lesson 5, for comparison:

Comp_LED-neg

Assembler / Compiler	Source code (lines)		Program memory (words)		Data memory (bytes)	
	12F629	16F506	12F629	16F506	12F629	16F506
Microchip MPASM	27	20	20	14	1	0
HI-TECH PICC-Lite	10	12	20	18	2	4
HI-TECH C PRO Lite	10	12	23	24	2	2

As usual, the C source code is significantly shorter than the assembler version. The PICC-Lite compiler generates particularly efficient code, being as small as the hand-written assembler version – so there is really no penalty for using C in this example.

Note that the code sizes are generally slightly larger for the 12F629, compared with the 16F506 (baseline) versions. This reflects the need for bank selection instructions in the midrange architecture.

Programmable Voltage Reference

As described in greater detail in midrange lesson 9, the PIC12F629 provides an internal voltage reference, which can be set to one of 32 available voltages, from 0 V to 0.72 × VDD.

It is controlled by the VRCON register, as follows:

The reference voltage is set by the VR<3:0> bits and VRR, which selects a high or low voltage range:

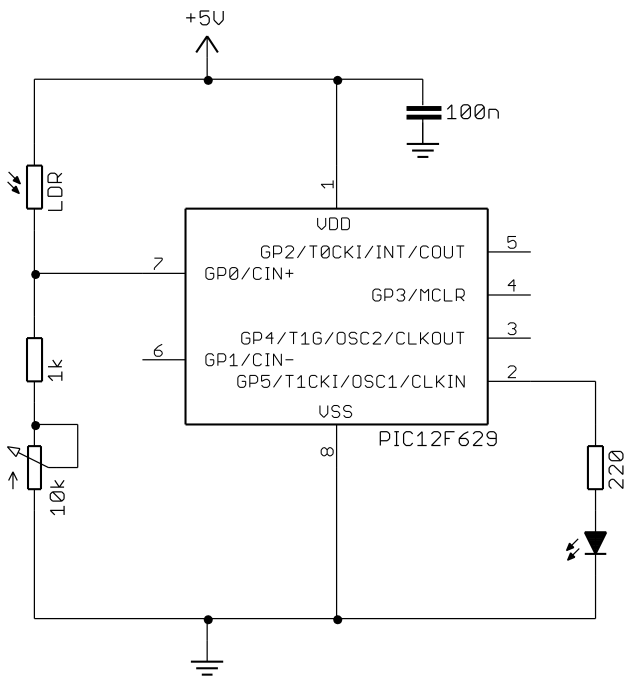
VRR = 1 selects the low range, where CVREF = VR<3:0>/24 × VDD.

VRR = 0 selects the high range, where CVREF = VDD/4 + VR<3:0>/32 × VDD.

With a 5 V supply, the available output range is from 0 V to 3.59 V.

Since the low and high ranges overlap, only 29 of the 32 selectable voltages are unique (0.250 × VDD, 0.500 × VDD and 0.625 × VDD are selectable in

The voltage reference module can be disabled (to save power in sleep mode) by clearing the VREN bit.



In midrange lesson 9, the circuit on the right was used to demonstrate how the voltage reference module can be used instead of an external reference – allowing us to eliminate the voltage divider from the previous example.

The CIN+ input is used with an LDR to detect light levels, as before. We'll use the LED on GP5 to indicate a low level of illumination, which we'll define to be CIN+ below 1.5 V.

On the PIC12F629, CVREF can only be used as the positive comparator reference. Since the signal from the LDR in our circuit is connected to CIN+, we have to configure CIN+ as the *negative* reference. This was done in midrange lesson 9 by:

```
        ; configure comparator
        movlw   b'110'|1<<CIS|0<<CINV
                                    ; select mode 6 (CM = 110):
                                    ;   +ref is CVref,
                                    ;   -ref is CIN+ (CIS = 1),
                                    ;   no external output,
                                    ;   comparator on
                                    ; output not inverted (CINV = 0)
        banksel CMCON               ;  -> COUT = 1 if CIN+ < CVref
        movwf   CMCON
```

To generate the closest match to 1.5 V, the voltage reference was configured with VRR = 1 (selecting the low range), and VR = 7, giving a threshold of 0.292 × 5.0 V = 1.46 V (when VDD = 5.0 V):

```
        ; configure voltage reference
        movlw   1<<VRR|.7|1<<VREN
                                    ; CVref = 0.292*Vdd (VRR = 1, VR = 7)
                                    ; enable voltage reference (VREN = 1)
        banksel VRCON               ; -> CVref = 1.5 V (if Vdd = 5.0 V)
        movwf   VRCON
```

HI-TECH C PRO or PICC-Lite

The comparator is configured in the same manner as in the last example:

```
    // configure comparator
    CMCON = 0b110;              // select mode 6 (CM = 110):
                                //   +ref is CVref,
                                //   no external output,
                                //   comparator on
    CIS = 1;                    // -ref is CIN+
    CINV = 0;                   // output not inverted
                                // -> COUT = 1 if CIN+ < CVref
```

Note that, this time, we need to explicitly set the CIS bit.

When configuring the voltage reference, we can use a similar "trick" to the comparator initialisation: since the VR bits are the lower four bits of VRCON, we can set the VR bits to a given value by simply assigning that value to `VRCON`:

```
    // configure voltage reference
    VRCON = 7;                  // select voltage: VR = 7,
    VRR = 1;                    //   low range -> CVref = 0.292*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.5 V (if Vdd = 5.0 V)
```

And, as with the comparator initialisation, the order of these statements is important. Since '`VRCON = 7`' clears the upper bits of VRCON, including VRR and VREN, you must initialise VRR or VREN after loading the VR value into VRCON, as shown.

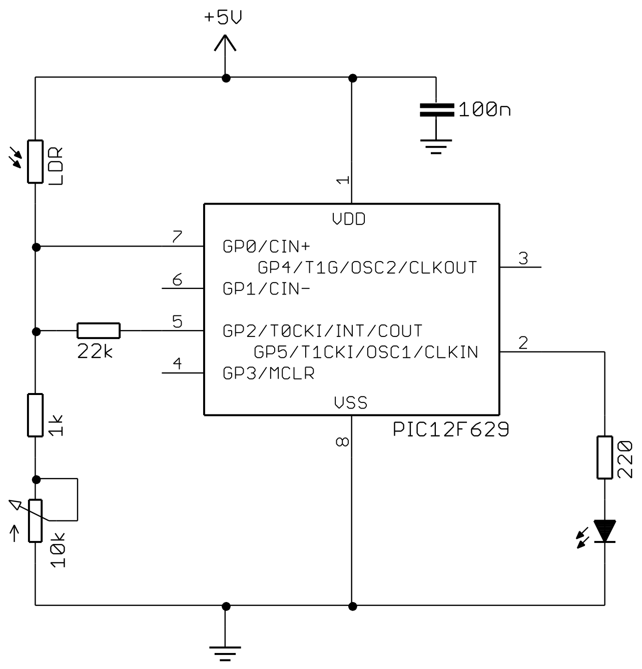
In the main loop, all we need do is continually copy the comparator output to the LED, as before:

```
    for (;;) {
        LED = COUT;             // continually display comparator output
    }
```

External Output and Hysteresis

As was explained in midrange lesson 9, it is often desirable to add hysteresis to a comparator, to make it less sensitive to small changes in the input signal due to superimposed noise or other interference. For example, in the above examples using an LDR, you will find that the output LED flickers when the light level is close to the threshold, particularly with mains-powered artificial illumination, which varies at 50 or 60 Hz.

On the PIC12F629, the comparator output can be made available on the COUT pin (shared with GP2); we can use this to add hysteresis to the circuit.



In the circuit on the right, hysteresis has been introduced by using a 22 kΩ resistor to feed some of the comparator's output, on COUT, back into the CIN+ input.

The comparator output is enabled in comparator modes 1, 3 and 5 (binary 001, 011 and 101).

Additionally, TRISIO<2> must be cleared to allow the COUT signal to appear on the GP2/COUT pin.

Hysteresis relies on positive feedback.

As we saw in the previous example, if you are using the internal voltage reference, your external signal must be connected to the comparator's negative input.

But that's not a problem – it is possible to introduce hysteresis by feeding the comparator output into the negative input signal (i.e. the signal being measured), as long as we invert the comparator output, so that the overall feedback is still positive.

HI-TECH C PRO or PICC-Lite

The code is essentially the same as before, but we must select comparator mode 5 (binary 101) instead of mode 6, to enable COUT:

```
    // configure comparator
    CMCON = 0b101;          // select mode 5 (CM = 101):
                            //   +ref is CVref,
                            //   external output enabled,
                            //   comparator on
    CIS = 1;                // -ref is CIN+
    CINV = 1;               // inverted output
                            // -> COUT = 1 if CIN+ > CVref,
                            //    COUT pin enabled
```

It is very important to set the CINV bit, so that the comparator output, fed via the 22 kΩ resistor into the comparator's negative input, will be inverted, providing positive feedback overall.

But before the comparator output can actually appear on the COUT pin, that pin has to be configured as an output:

```
    TRIS2 = 0;              // configure COUT (GP2) as an output
```

Note that we could have done this when initialising the port, for example:

```
    TRISIO = ~(1<<nLED|1<<2);    // configure LED pin and COUT (GP2) as outputs
```

However, it is usually better to keep all code associated with the comparator configuration (such as enabling the COUT pin) with the rest of the comparator initialisation code; if you later want to change how the comparator is configured, you don't have to remember to make a corresponding in change in some other section of initialisation code. Modular code tends to be more easily maintained.

Finally, because the comparator output is now inverted, we need to invert the display:

```
    for (;;)
    {
        LED = !COUT;         // continually display inverse of comparator output
    }
```

Complete program

Here is how these pieces fit together:

```
*************************************************************************
*                                                                       *
*   Description:    Lesson 6, example 3                                 *
*                                                                       *
*   Demonstrates comparator hysteresis (using COUT)                     *
*   with programmable voltage reference                                 *
*                                                                       *
*   Turns on LED when voltage on CIN+ < 1.5 V                           *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       CIN+  - voltage to be measured (e.g. pot output or LDR)         *
*       COUT  - comparator output (fed back to input via resistor)      *
*       GP5   - indicator LED                                           *
*                                                                       *
************************************************************************/

#include <htc.h>


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

// Pin assignments
#define LED     GPIO5           // indicator LED on GP5
#define nLED    5               //   (port bit 3)


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

    // configure ports
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as an output

    // configure comparator
    CMCON = 0b101;              // select mode 5 (CM = 101):
                                //   +ref is CVref,
                                //   external output enabled,
                                //   comparator on
    CIS = 1;                    // -ref is CIN+
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if CIN+ > CVref,
                                //    COUT pin enabled

    TRIS2 = 0;                  // configure COUT (GP2) as an output

    // configure voltage reference
    VRCON = 7;                  // select voltage: VR = 7,
    VRR = 1;                    //   low range -> CVref = 0.292*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.5 V (if Vdd = 5.0 V)

    // Main loop
    for (;;)
    {
        LED = !COUT;        // continually display inverse of comparator output
    }
}
```

Comparator Interrupts

The comparator module can be selected as an interrupt source. This allows us to respond immediately to a change in the comparator output (i.e. the analog input crossing a threshold), without having to continually poll COUT, as we were doing above. Instead, you can choose to generate an interrupt whenever COUT changes, and handle the "comparator change" event in your interrupt service routine (ISR) – much as we did for pin change interrupts in lesson 4.

Like all interrupts (see lesson 3), the comparator interrupt has an associated enable bit, which has to be set to allow the comparator to trigger interrupts, and an interrupt flag bit, which is set whenever the comparator's output changes. As we saw in midrange lesson 9, the comparator is considered to be a *peripheral* (not part of the PIC core), and its interrupt bits are held in peripheral interrupt registers.

Peripheral interrupts on the PIC12F629 are enabled by setting the PEIE (peripheral interrupt enable) bit in the INTCON register.

The comparator interrupt is enabled by setting the CMIE bit in the PIE1 (peripheral interrupt enable 1) register.

And as we have seen, the GIE bit, in the INTCON register, must be set to globally enable interrupts.

So, to enable comparator interrupts, we must set CMIE, PEIE and GIE.

The comparator interrupt flag, CMIF, is located in PIR1 (peripheral interrupt register 1). It is set whenever the comparator output (COUT) has changed since the last time CMCON was read or written.

As with other interrupt sources, the comparator interrupt flag is active, whether or not comparator interrupts are enabled. This means that you could, in principle, poll CMIF to detect changes in COUT – although it would normally make more sense to simply poll COUT directly.

In the interrupt service routine, you should read or write CMCON, to clear any existing *mismatch condition*, to allow you to successfully clear the CMIF flag. And to avoid false triggering (an interrupt occurring because of a previous change), you should also read or write CMCON and clear CMIF, immediately before enabling the comparator interrupt.

To demonstrate this, we can re-implement the last example, using interrupts.

Instead of polling COUT, we'll set the comparator module to trigger an interrupt whenever the input (light level) passes through the threshold, and have the ISR turn on or off the LED.

This then leaves the main program code free to perform other tasks – although in this example, all we will do is update the LED display.

We can keep the comparator and voltage reference configuration the same as before. It's a good idea to leave the hysteresis in place, as this minimises the number of comparator transitions – limiting the number of times the interrupt will be triggered.

In midrange lesson 9, the comparator interrupt was enabled by:

```
        movlw   1<<PEIE|1<<GIE      ; enable peripheral and global interrupts
        movwf   INTCON
        banksel CMCON               ; enable comparator interrupt:
        movf    CMCON,w             ;   read CMCON to clear mismatch
        banksel PIR1
        bcf     PIR1,CMIF           ;   clear interrupt flag
        banksel PIE1
        bsf     PIE1,CMIE           ;   set enable bit
```

In the interrupt handler, the comparator mismatch condition was removed first, allowing the interrupt flag to be cleared:

```
        banksel CMCON
        movf    CMCON,w             ; clear mismatch condition
        banksel PIR1
        bcf     PIR1,CMIF           ; clear interrupt flag
```

Since the comparator interrupt is triggered by any comparator output change, we can't know whether the interrupt was caused by the comparator input going high or low.

So, we checked the value of COUT, and set the LED output accordingly, using:

```
        ; test for low comparator input and display
        clrf    sGPIO               ; assume COUT = 0 -> LED off
        banksel CMCON
        btfss   CMCON,COUT          ; if comparator output low (CIN+ < 1.5 V)
        bsf     sGPIO,nLED          ;   turn on LED (using shadow register)
```

The main loop then simply copied the shadow register to GPIO.

HI-TECH C PRO or PICC-Lite

Implementing these steps using HI-TECH C is quite straightforward; the techniques are similar to those used to in the interrupt-on-change example in lesson 4.

First we enable the peripheral and global interrupts:

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

Recall that, before enabling the comparator interrupt, we should read CMCON, and clear CMIF, in case a comparator change has occurred since execution began, to avoid triggering the comparator interrupt the moment it is enabled[4].

We saw in lesson 4 that, given an expression consisting only of a register name, both HI-TECH C compilers will generate an instruction which does nothing other than read that register.

So, to read CMCON, we can simply write:

```
CMCON;                          //   read CMCON to clear mismatch
```

and then clear CMIF and enable the comparator interrupt:

```
CMIF = 0;                       //   clear interrupt flag
CMIE = 1;                       //   set enable bit
```

The interrupt initialisation is done in this order, with peripheral and global interrupts being enabled first, to limit the time between clearing CMCON and enabling the comparator interrupt. But of course, a few microseconds won't really matter, so in practice you could instead enable the comparator interrupt (and any other interrupt sources) before enabling peripheral and global interrupts last:

```
                                // enable comparator interrupt:
CMCON;                          //   read CMCON to clear mismatch
CMIF = 0;                       //   clear interrupt flag
CMIE = 1;                       //   set enable bit
PEIE = 1;                       // enable peripheral interrupts
ei();                           // enable global interrupts
```

If you take this approach and enable global interrupts last, all your interrupt sources will become active at once. This may or may not represent a problem, but it's something to consider. In this instance, with only one interrupt source enabled, it makes no real difference.

In the interrupt handler, we explicitly clear the comparator mismatch (instead of relying on later statements within the handler routine), and clear the interrupt flag:

```
CMCON;                          // read CMCON to clear mismatch condition
CMIF = 0;                       // clear interrupt flag
```

We can then turn the LED on or off, depending on whether the comparator output has gone high or low.

In the assembler example, this was done by first clearing the shadow register and then setting the bit corresponding to the LED only if COUT = 0 (implying that CIN+ < 1.5 V). But in 'C', this is more naturally expressed as:

```
if (COUT)                       // if comparator output high (CIN+ > 1.5 V)
{
    sGPIO &= ~(1<<nLED);        //   turn off LED
}
else                            // else (CIN+ < 1.5 V)
{
    sGPIO |= 1<<nLED;           //   turn on LED
}
```

[4] In this example, there would be no problem if the comparator interrupt was triggered by a change which had happened before it was enabled, but that may not be true in other applications.

Then in the main loop, we do nothing but copy the shadow register to the port:

```
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO;
    }
```

Complete program

Here is how these code fragments come together:

```
/************************************************************************
*   Description:    Lesson 6, example 4                                 *
*                                                                       *
*   Demonstrates use of comparator interrupt                            *
*   (assumes hysteresis is used to reduce triggering)                   *
*                                                                       *
*   Turns on LED when voltage on CIN+ < 1.5 V                           *
*                                                                       *
*************************************************************************
*   Pin assignments:                                                    *
*       CIN+  - voltage to be measured (e.g. pot output or LDR)         *
*       COUT  - comparator output (fed back to input via resistor)      *
*       GP5   - indicator LED                                           *
*                                                                       *
************************************************************************/

#include <htc.h>


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

// Pin assignments
#define LED     GPIO5           // indicator LED on GP5
#define nLED    5               //   (port bit 5)


/***** GLOBAL VARIABLES *****/
unsigned char   sGPIO;          // shadow copy of GPIO


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

    TRISIO = ~(1<<nLED);        // configure LED pin (only) as an output

    // configure comparator
    CMCON = 0b101;              // select mode 5 (CM = 101):
                                //   +ref is CVref,
                                //   external output enabled,
                                //   comparator on
    CIS = 1;                    // -ref is CIN+
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if CIN+ > CVref,
                                //    COUT pin enabled
    TRIS2 = 0;                  // configure COUT (GP2) as an output

    // configure voltage reference
    VRCON = 7;                  // select voltage: VR = 7,
    VRR = 1;                    //   low range -> CVref = 0.292*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.5 V (if Vdd = 5.0 V)

    // configure interrupts
    PEIE = 1;                   // enable peripheral interrupts
    ei();                       // enable global interrupts
                                // enable comparator interrupt:
    CMCON;                      //   read CMCON to clear mismatch
    CMIF = 0;                   //   clear interrupt flag
    CMIE = 1;                   //   set enable bit

    // Main loop
    for (;;)
    {
        // continually copy shadow GPIO to port
        GPIO = sGPIO;
    }
}


/***** INTERRUPT SERVICE ROUTINE *****/
void interrupt isr(void)
{
    // Service comparator interrupt
    //   Triggered on any comparator output change,
    //   caused by comparator input crossing 1.5 V threshold
    //
    CMCON;                      // read CMCON to clear mismatch condition
    CMIF = 0;                   // clear interrupt flag

    // turn on LED if CIN+ < 1.5 V (using shadow register)
    if (COUT)                   // if comparator output high (CIN+ > 1.5 V)
    {
        sGPIO &= ~(1<<nLED);    //   turn off LED
    }
    else                        // else (CIN+ < 1.5 V)
    {
        sGPIO |= 1<<nLED;       //   turn on LED
    }
}
```

Comparisons

Here the resource usage summary for this example:

Comp_Interrupt

Assembler / Compiler	**Source code (lines)**	**Program memory (words)**	**Data memory (bytes)**
Microchip MPASM	60	46	3
HI-TECH PICC-Lite	28	51	5
HI-TECH C PRO Lite	28	79	7

Once again, the C source code is less than half as long as the assembler source, reflecting the extent to which HI-TECH C is able to make some of the details of implementing interrupts transparent, while the PICC-Lite compiler generated optimised code only 10% larger than the hand-written assembly version.

Wake-up on Comparator Change

The comparator module does not depend on the PIC's oscillator, and so can remain active while the device is in sleep mode[5]. This means that the comparator can be used to wake the PIC from sleep mode, as can any interrupt source, which is active in sleep mode – as explained in lesson 4 and in midrange lesson 9.

This is useful for conserving power while waiting for a signal change from a sensor.

To enable wake-up on comparator change, simply enable the comparator interrupt, by setting the CMIE and PEIE enable bits.

If you only want to use the comparator to wake the PIC from sleep mode, and do not actually want to use interrupts, leave interrupts disabled, by clearing GIE.

This was demonstrated in midrange lesson 9, using the previous circuit, by configuring the PIC to wake on comparator change, and then go to sleep. When the input signal crosses the threshold level, the comparator output changes, and the PIC wakes. To indicate this we turned on the LED for one second. The PIC then went back to sleep, to wait until the next comparator change.

In the assembler example, after configuring the comparator and voltage reference as usual, a delay of 10 ms was added to allow them to settle before entering sleep mode. This is necessary because signal levels can take a while to settle after power-on, and if the comparator output was changing while going into sleep mode, the PIC would immediately wake and the LED would flash – a false trigger.

HI-TECH PICC-Lite

To enable wake on comparator change, we need to enable the comparator interrupt, but not global interrupts (since we're not actually using interrupts here; we're only using an interrupt source to wake the device):

```
    // enable comparator interrupt (for wake on change)
    PEIE = 1;                   // enable peripheral interrupts
    CMIE = 1;                   // enable comparator interrupt
```

Within the main loop, we can start by turning off the LED, because we don't want it lit on start-up:

```
        // turn off LED
        LED = 0;
```

Before entering sleep mode, it is very important to clear any existing comparator mismatch condition, as well as CMIF:

```
        // enter sleep mode
        CMCON;                  // read CMCON to clear comparator mismatch
        CMIF = 0;               // clear comparator interrupt flag
        SLEEP();
```

[5] If you are not using the comparator to wake the device, you should turn it off before entering sleep mode, to conserve power.

The PIC will now sleep until it is woken by a comparator change, which we wish to show by lighting the LED for one second:

```
        // turn on LED for 1 second
        LED = 1;                // turn on LED
        DelayS(1);              // delay 1 sec
```

Note that the delay is generated by the `DelayS()` macro, introduced in lesson 2.

Complete program

Here is the complete PICC-Lite version of the comparator wakeup example program:

```
/************************************************************************
*   Description:    Lesson 6, example 5                                 *
*                                                                       *
*   Demonstrates wake-up on comparator change                           *
*                                                                       *
*   Turns on LED for 1 sec when comparator output changes               *
*   then sleeps until the next change                                   *
*   (threshold is 1.5 V, set by programmable voltage ref),              *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       CIN+  - voltage to be measured (e.g. pot output or LDR)         *
*       GP5   - indicator LED                                           *
*                                                                       *
************************************************************************/

#include <htc.h>

#define XTAL_FREQ   4MHZ     // oscillator frequency for DelayMs()
#include "stdmacros-PCL.h"   // defines DelayS()


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

// Pin assignments
#define LED     GPIO5           // indicator LED on GP5
#define nLED    5               //   (port bit 5)


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

    // configure ports
    TRISIO = ~(1<<nLED);        // configure LED pin (only) as an output

    // configure comparator
    CMCON = 0b110;              // select mode 6 (CM = 110):
                                //   +ref is CVref,
                                //   no external output,
                                //   comparator on
    CIS = 1;                    // -ref is CIN+
    CINV = 0;                   // output not inverted
                                // -> COUT = 1 if CIN+ < CVref
    // configure voltage reference
    VRCON = 7;                  // select voltage: VR = 7,
    VRR = 1;                    //   low range -> CVref = 0.292*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.5 V (if Vdd = 5.0 V)

    // delay 10 ms to allow comparator and voltage ref to settle
    DelayMs(10);

    // enable comparator interrupt (for wake on change)
    PEIE = 1;                   // enable peripheral interrupts
    CMIE = 1;                   // enable comparator interrupt


    // Main loop
    for (;;)
    {
        // turn off LED
        LED = 0;

        // enter sleep mode
        CMCON;                  // read CMCON to clear comparator mismatch
        CMIF = 0;               // clear comparator interrupt flag
        SLEEP();

        // turn on LED for 1 second
        LED = 1;                // turn on LED
        DelayS(1);              // delay 1 sec
     }
}
```

HI-TECH C PRO

To adapt this code for HI-TECH C PRO, very little change is needed – most of the code is the same, as usual.

But because HI-TECH C PRO provides built-in delay functions, which are a little different from the example delay functions provides with PICC-Lite, we need to change the oscillator frequency definition from:

```
#define XTAL_FREQ   4MHZ        // oscillator frequency for delay functions
```

to:

```
#define _XTAL_FREQ  4000000     // oscillator frequency for delay functions
```

and change the 10 ms delay from '`DelayMs(10)`' to '`__delay_ms(10)`'.

We also need to include the appropriate header file, defining the '`DelayS()`' macro (from lesson 2):

```
#include "stdmacros-HTC.h"      // DelayS() - delay in seconds
```

instead of:

```
#include "stdmacros-PCL.h"      // DelayS() - delay in seconds
```

so that the appropriate millisecond delay function will be called.

Comparisons

Here is the resource usage for the "wake-up on comparator change demo" assembler and C examples, along with the baseline (PIC16F506) versions of this example, from baseline C lesson 5, for comparison:

Comp_Wakeup

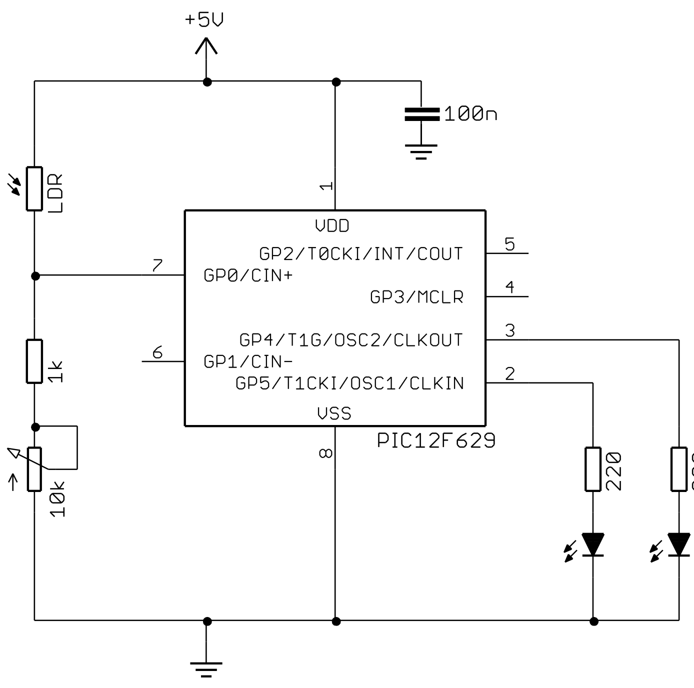
Assembler / Compiler	**Source code (lines)**		**Program memory (words)**		**Data memory (bytes)**	
	12F629	16F506	12F629	16F506	12F629	16F506
Microchip MPASM	38	31	43	42	3	3
HI-TECH PICC-Lite	24	22	57	77	6	10
HI-TECH C PRO Lite	24	22	66	63	5	4

Note that the PICC-Lite compiler generated much smaller code for the midrange architecture than the baseline version, presumably because the larger stack available on midrange PICs makes it easier to optimise the HI-TECH provided delay code.

Otherwise, the pattern is much the same as we have seen before – the C source code being significantly smaller than the assembler version, but the C compilers generating significantly larger code than hand-written assembler.

Comparing an input against a range of thresholds

You may need to check that a signal is within certain limits and have to take action if the signal is too high or too low. Or there may be a number of thresholds you need to compare the signal against.



The circuit on the right was used in midrange lesson 9 to demonstrate how the programmable voltage reference can be used to achieve this. The LED on GP5 is used to indicate a low level of illumination, while the LED on GP4 indicates bright light. When neither LED is lit, the light level will be in the middle; not too dim or too bright.

To test whether the input is within limits, the programmable voltage reference is first configured to generate the "low" threshold voltage, the input is compared with this low level, and then the voltage reference reconfigured to generate the "high" threshold and the input compared with this higher level.

This process could be extended to multiple input thresholds, by configuring the voltage reference to generate each threshold in turn. However, if you wish to test against more than a few threshold levels, you would probably be better off using an analog-to-digital converter (described in lesson 12).

This example uses 1.0 V as the "low" threshold and 2.0 V as the "high" threshold, but, since the reference is programmable, you can always choose your own levels!

HI-TECH C PRO or PICC-Lite

The comparator is configured to compare CIN+ with CVREF, as we have done before:

```
    CMCON = 0b110;              // select mode 6 (CM = 110):
                                //   +ref is CVref,
                                //   no external output,
                                //   comparator on
    CIS = 1;                    // -ref is CIN+
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if CIN+ > CVref
```

The voltage reference can be configured to generate approximately 1.0 V, by:

```
    VRCON = 5;                  // select voltage: VR = 5,
    VRR = 1;                    //   low range -> CVref = 0.208*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.04 V (if Vdd = 5.0 V)
```

The closest match to 2.0 V is obtained by:

```
    VRCON = 5;                  // select voltage: VR = 5,
    VRR = 0;                    //   high range -> CVref = 0.406*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 2.03 V (if Vdd = 5.0 V)
```

After changing the voltage reference, it can take a little while for it to settle and generate a stable voltage. According to table 12-7 in the PIC12F629/675 data sheet, this settling time can be up to 10 µs.

Therefore, we should insert a 10 µs delay after configuring the voltage reference, before reading the comparator output.

This can be done using the '`DelayUs()`' macro defined in the "delay.h" header provided with PICC-Lite:

```
    DelayUs(10);                // wait 10 us to settle
```

or the '`__delay_us()`' macro built into HI-TECH C Pro:

```
    __delay_us(10);             // wait 10 us to settle
```

Complete program

Here is the complete HI-TECH C PRO version of the "test that a signal is within limits" example:

```
/************************************************************************
*   Description:    Lesson 6, example 6                                 *
*                                                                       *
*   Demonstrates use of programmable voltage reference                  *
*   to test that a signal is within limits                              *
*                                                                       *
*   Turns on Low LED  when CIN+ < 1.0 V (low light level)               *
*         or High LED when CIN+ > 2.0 V (high light level)              *
*                                                                       *
*************************************************************************
*   Pin assignments:                                                    *
*       CIN+  - voltage to be measured (e.g. pot output or LDR)         *
*       GP5   - "Low" LED                                               *
*       GP4   - "High" LED                                                 *
*                                                                          *
************************************************************************/

#include <htc.h>

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


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

// Pin assignments
#define nLO     5               // "Low" LED on GP5
#define nHI     4               // "High" LED on GP4


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

    // Initialisation
    TRISIO = ~(1<<nLO|1<<nHI);  // configure LED pins as outputs

    // configure comparator
    CMCON = 0b110;              // select mode 6 (CM = 110):
                                //   +ref is CVref,
                                //   no external output,
                                //   comparator on
    CIS = 1;                    // -ref is CIN+
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if CIN+ > CVref


    // Main loop
    for (;;)
    {
        sGPIO = 0;                  // start with both LEDs off

        // Test for low illumination
        // set low input threshold
        VRCON = 5;                  // select voltage: VR = 5,
        VRR = 1;                    //   low range -> CVref = 0.208*Vdd
        VREN = 1;                   // enable voltage reference
                                    // -> CVref = 1.04 V (if Vdd = 5.0 V)
        __delay_us(10);             // wait 10us to settle
        // compare with input
        if (!COUT)                  // if CIN+ < CVref
            sGPIO |= 1<<nLO;        //   turn on Low LED

        // Test for high illumination
        // set high input threshold
        VRCON = 5;                  // select voltage: VR = 5,
        VRR = 0;                    //   high range -> CVref = 0.406*Vdd
        VREN = 1;                   // enable voltage reference
                                    // -> CVref = 2.03 V (if Vdd = 5.0 V)
        __delay_us(10);             // wait 10us to settle
        // compare with input
        if (COUT)                   // if CIN+ > CVref
            sGPIO |= 1<<nHI;        //   turn on High LED

        // Display test results
        GPIO = sGPIO;               // copy shadow to GPIO port
    }
}
```

Testing two independent inputs

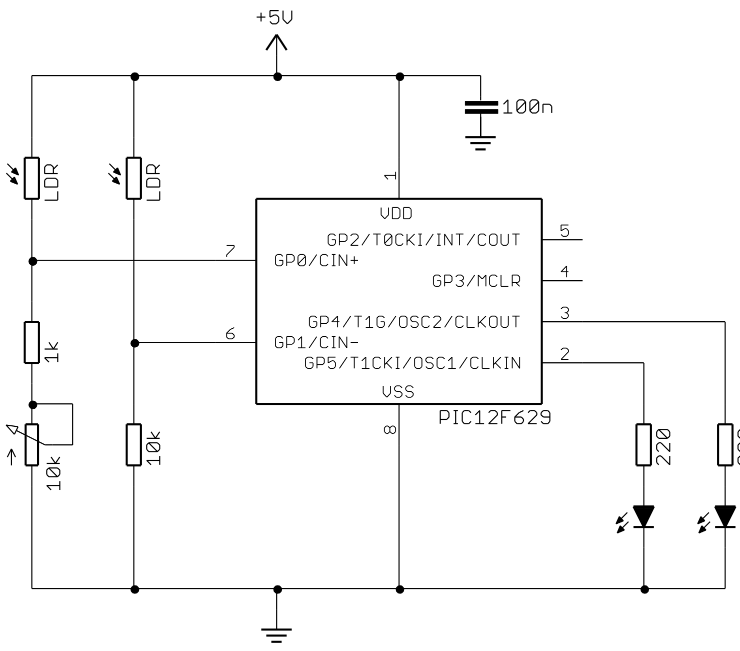
Suppose that you need to compare two input signals (say, light level in two locations) against a common reference. You could use two comparators, with each input connected to a different comparator, and the common reference voltage connected to both. But, what if you only have a single comparator available, as in the PIC12F629?

Recall that, in comparator modes 5 and 6 (binary 101 and 110), the CIS bit in the CMCON register selects whether the CIN+ or CIN- pin is connected to the comparator's negative input.

In these modes, the negative input is said to be *multiplexed*, making it easy to switch between the CIN+ and CIN- pins, on the fly.

This means that we can select CIN+ as an input, perform a comparison with the internal reference voltage, and then select CIN- and repeat the comparison[6].

This was demonstrated in midrange lesson 9, using the circuit on the right, where an additional LDR was added to the circuit used in the last example. The LED on GP4 was lit whenever the CIN+ input is above 1.5 V (indicating bright light), while the LED on GP5 indicates bright light on the LDR connected to CIN-.



HI-TECH C PRO or PICC-Lite

When configuring the comparator, there is no need to explicitly initialise CIS, since we'll be using it to switch between inputs, later:

```
    CMCON = 0b110;              // select mode 6 (CM = 110):
                                //   +ref is CVref,
                                //   -ref is CIN+ or CIN- (selected by CIS)
                                //   no external output,
                                //   comparator on
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if -ref > CVref
```

[6] Since the voltage reference is programmable, you could setup a different threshold for each input, by reprogramming the reference voltage each time you switch between inputs – but for clarity we won't do that in this example.

Within the main loop, we can then select each input in turn:

```
        // test input 1
        CIS = 1;                    // select CIN+ pin as –ref
```

After changing the comparator's configuration, or mode, it can take a while for it to settle and generate a valid output. According to table 12-6 in the PIC629/675 data sheet, this ("Comparator Mode Change to Output Valid") can take up to 10 µs. By coincidence, that's the same as the voltage reference settling time, discussed in the last example, although they are actually unrelated.

Therefore, we need to add a 10 µs delay after selecting a new input, before reading the comparator output, using:

```
        DelayUs(10);                // wait 10us to settle
```

for PICC-Lite or

```
        __delay_us(10);             // wait 10us to settle
```

for HI-TECH C PRO.

We can then test the comparator output, and turn on the corresponding LED if the CIN+ input is higher than the threshold:

```
        if (COUT)                   // if CIN+ > CVref
            sGPIO |= 1<<nLED1;      //   turn on LED 1
```

We can then do the same again, selecting the other comparator input:

```
        // test input 2
        CIS = 0;                    // select CIN- pin as -ref
        __delay_us(10);             // wait 10us to settle
        if (COUT)                   // if CIN- > CVref
            sGPIO |= 1<<nLED2;      //   turn on LED 2
```

And, since this code has updated a shadow variable, instead of writing directly to GPIO, we need to display any updates by writing the shadow copy to GPIO:

```
        // display test results
        GPIO = sGPIO;               // copy shadow to GPIO port
```

Complete program

Here is the full PICC-Lite version of the "two inputs with a common programmed voltage reference" program, showing how these fragments fit together:

```
*************************************************************************
*   Description:    Lesson 6, example 7                                 *
*                                                                       *
*   Demonstrates comparator input multiplexing                          *
*                                                                       *
*   Turns on: LED 1 when CIN+ > 1.5 V                                   *
*         and LED 2 when CIN- > 1.5 V                                   *
*                                                                       *
*************************************************************************
*                                                                       *
*   Pin assignments:                                                    *
*       CIN+  - input 1 (LDR/resistor divider)                          *
*       CIN-  - input 2 (LDR/resistor divider)                          *
*       GP4   - indicator LED 1                                         *
*       GP5   - indicator LED 2                                         *
*                                                                       *
************************************************************************/

#include <htc.h>

#define XTAL_FREQ   4MHZ    // oscillator frequency for DelayUs()
#include "delay.h"          // defines DelayUs()


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

// Pin assignments
#define nLED1   4               // indicator LED 1 on GP4
#define nLED2   5               // indicator LED 2 on GP5


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

    // Initialisation
    TRISIO = ~(1<<nLED1|1<<nLED2);  // configure LED pins as outputs

    // configure comparator
    CMCON = 0b110;              // select mode 6 (CM = 110):
                                //   +ref is CVref,
                                //   -ref is CIN+ or CIN- (selected by CIS)
                                //   no external output,
                                //   comparator on
    CINV = 1;                   // inverted output
                                // -> COUT = 1 if -ref > CVref

    // configure voltage reference
    VRCON = 7;                  // select voltage: VR = 7,
    VRR = 1;                    //   low range -> CVref = 0.292*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.5 V (if Vdd = 5.0 V)


    // Main loop
    for (;;)
    {
        sGPIO = 0;                  // start with both LEDs off

        // test input 1
        CIS = 1;                    // select CIN+ pin as -ref
        DelayUs(10);                // wait 10us to settle
        if (COUT)                   // if CIN+ > CVref
            sGPIO |= 1<<nLED1;      //   turn on LED 1

        // test input 2
        CIS = 0;                    // select CIN- pin as -ref
        DelayUs(10);                // wait 10us to settle
        if (COUT)                   // if CIN- > CVref
            sGPIO |= 1<<nLED2;      //   turn on LED 2
        // display test results
        GPIO = sGPIO;               // copy shadow to GPIO port
    }
}
```

Comparisons

Here the resource usage summary for this example:

Comp_2input

Assembler / Compiler	**Source code (lines)**	**Program memory (words)**	**Data memory (bytes)**
Microchip MPASM	45	38	1
HI-TECH PICC-Lite	25	39	4
HI-TECH C PRO Lite	24	50	4

The C source code is still only around half as long as the assembler source, while the PICC-Lite compiler was able to generate highly optimised code; barely any larger than the hand-written assembly version.

Testing four independent inputs

If you need more than two comparator inputs, you could either use an external device (an analog switch, or multiplexer) to switch between multiple signals, or you could simply use a PIC with more than a single comparator.

Introducing the PIC16F684

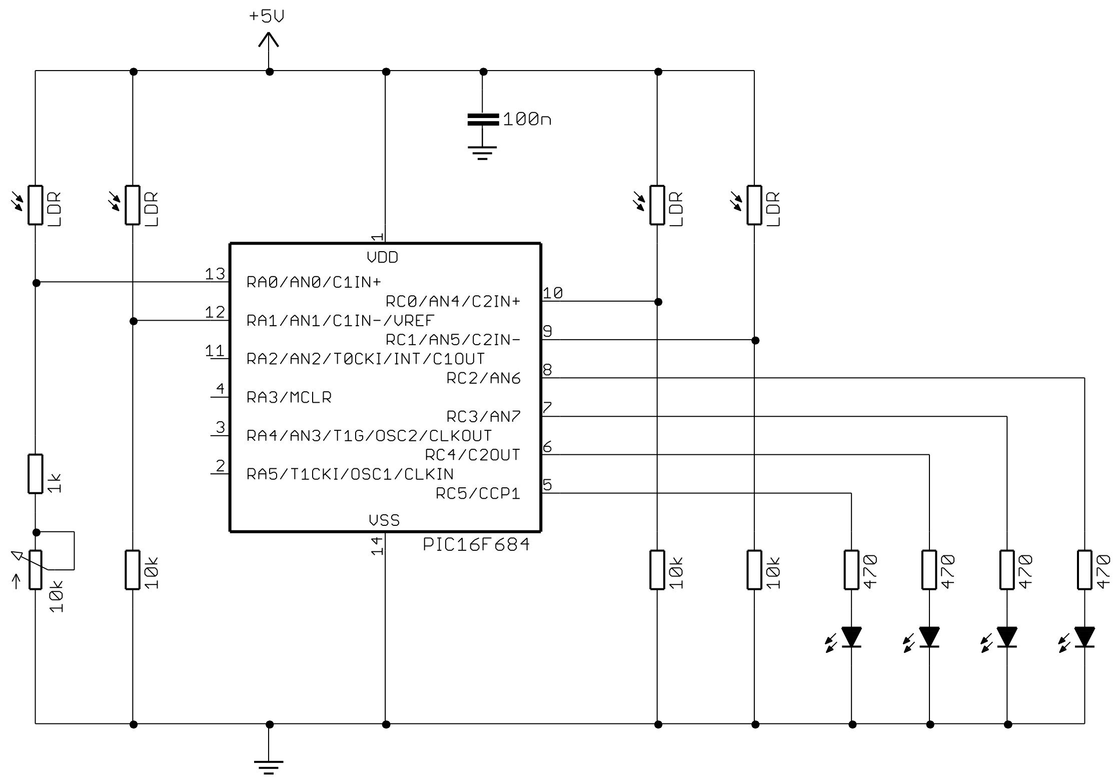
The PIC16F684, introduced in midrange lesson 10, includes a dual comparator module that is essentially an extended version of the PIC16F629's single comparator; see midrange lesson 11 for details.

One reason the 16F684 was chosen for this tutorial series is that, unlike say the 16F688, the 16F684 is supported by HI-TECH's PICC-Lite compiler, with full optimisation. This allows us to properly compare the assembler examples from midrange lesson 11 with optimised C equivalents.

However, although the PICC-Lite compiler generates optimised code for the 16F684, it is limited. Unlike the 12F629, PICC-Lite does not allow us to use all the memory on the 16F684, limiting program memory use to 1024 words (out of the 2048 words available in the device) and data memory to 96 bytes (out of 128 bytes in total; PICC-Lite limits general purpose register use to bank 0).

Thus, although HI-TECH C PRO, in "Lite" mode, may generate unoptimised code up to twice the size of that generated by PICC-Lite, you may actually be able to write a longer, more complex program for the 16F684 using HI-TECH C PRO, which does not impose any memory usage restrictions.

To demonstrate how the 16F684's dual comparator module can be used with four inputs, we'll use the circuit with four LDRs shown on the next page.



As in the last example, the programmable voltage reference is used to generate a common threshold level of 1.5 V. When any of the comparator inputs is higher than this threshold (corresponding to bright light), the corresponding LED is turned on.

HI-TECH C PRO or PICC-Lite

To use the programmable voltage reference on the 16F684, we must choose comparator mode 2 (010).

As in the last example, when configuring the comparator, there is no need to initialise CIS, since we'll be using it to switch between inputs, later:

```
    // configure comparators
    CMCON0 = 0b010;           // select mode 2 (CM = 010):
                              //   C1 +ref is CVref,
                              //   C1 -ref is C1IN+ or C1IN- (selected by CIS),
                              //   C2 +ref is CVref,
                              //   C2 -ref is C2IN+ or C2IN- (selected by CIS),
                              //   no external outputs,
                              //   both comparators on
    C1INV = 1;                // C1 output inverted
                              // -> C1OUT = 1 if C1 -ref > CVref
    C2INV = 1;                // C2 output inverted
                              // -> C2OUT = 1 if C2 -ref > CVref
```

The voltage reference is configured as before:

```
    // configure voltage reference
    VRCON = 7;                  // select voltage: VR = 7,
    VRR = 1;                    //   low range -> CVref = 0.292*Vdd
    VREN = 1;                   // enable voltage reference
                                // -> CVref = 1.5 V (if Vdd = 5.0 V)
```

The tests in the main loop are much the same as in the previous example, except that changing the value of CIS now selects a pair of inputs, instead of a single input, because it affects both comparators:

```
        // test inputs 1 and 3
        CIS = 1;                    // select C1IN+ and C2IN+ pins as -refs
        DelayUs(10);                // wait 10us to settle
        if (C1OUT)                  // if C1IN+ > CVref
            sPORTC |= 1<<nLED1;     //   turn on LED 1
        if (C2OUT)                  // if C2IN+ > CVref
            sPORTC |= 1<<nLED3;     //   turn on LED 3

        // test inputs 2 and 4
        CIS = 0;                    // select C1IN- and C2IN- pins as -refs
        DelayUs(10);                // wait 10us to settle
        if (C1OUT)                  // if C1IN- > CVref
            sPORTC |= 1<<nLED2;     //   turn on LED 2
        if (C2OUT)                  // if C1IN- > CVref
            sPORTC |= 1<<nLED4;     //   turn on LED 4
```

This the PICC-Lite version of the code, using the '`DelayUs()`' to add the 10 µs delay necessary to allow the comparators to settle after changing CIS.

For HI-TECH C PRO, use:

```
        __delay_us(10);             // wait 10us to settle
```

as in the last example.

Complete program

The structure of this "four inputs with a common programmed voltage reference" example is very similar to the "two inputs" program in the last example, but here is the full HI-TECH C version, for reference:

```
************************************************************************
*                                                                      *
*   Description:    Lesson 6, example 8                                *
*                                                                      *
*   Demonstrates dual comparator input multiplexing                    *
*                                                                      *
*   Turns on: LED 1 when C1IN+ > 1.5 V,                                *
*             LED 2 when C1IN- > 1.5 V,                                *
*             LED 3 when C2IN+ > 1.5 V,                                *
*         and LED 4 when C2IN- > 1.5 V                                 *
*                                                                      *
************************************************************************
*                                                                      *
*   Pin assignments:                                                   *
*       C1IN+ - input 1 (LDR/resistor divider)                         *
*       C1IN- - input 2 (LDR/resistor divider)                         *
*       C2IN+ - input 3 (LDR/resistor divider)                         *
*       C2IN- - input 4 (LDR/resistor divider)                         *
*       RC2   - indicator LED 1                                        *
*       RC3   - indicator LED 2                                        *
*       RC4   - indicator LED 3                                        *
*       RC5   - indicator LED 4                                        *
*                                                                      *
***********************************************************************/

#include <htc.h>

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

```
/***** 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 nLED1   2              // indicator LED 1 on RC2
#define nLED2   3              // indicator LED 2 on RC3
#define nLED3   4              // indicator LED 3 on RC4
#define nLED4   5              // indicator LED 4 on RC5


/***** MAIN PROGRAM *****/
void main()
{
    unsigned char   sPORTC;   // shadow copy of PORTC

    // Initialisation
    TRISC = 0b000011;         // configure RC0/C2IN+ and RC1/C2IN- as inputs,
                              //   rest of PORTC as outputs
                              //   (PORTA all inputs by default)

    // configure comparators
    CMCON0 = 0b010;           // select mode 2 (CM = 010):
                              //   C1 +ref is CVref,
                              //   C1 -ref is C1IN+ or C1IN- (selected by CIS),
                              //   C2 +ref is CVref,
                              //   C2 -ref is C2IN+ or C2IN- (selected by CIS),
                              //   no external outputs,
                              //   both comparators on
    C1INV = 1;                // C1 output inverted
                              // -> C1OUT = 1 if C1 -ref > CVref
    C2INV = 1;                // C2 output inverted
                              // -> C2OUT = 1 if C2 -ref > CVref

    // configure voltage reference
    VRCON = 7;                // select voltage: VR = 7,
    VRR = 1;                  //   low range -> CVref = 0.292*Vdd
    VREN = 1;                 // enable voltage reference
                              // -> CVref = 1.5 V (if Vdd = 5.0 V)


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

        // test inputs 1 and 3
        CIS = 1;                   // select C1IN+ and C2IN+ pins as -refs
        __delay_us(10);            // wait 10us to settle
        if (C1OUT)                 // if C1IN+ > CVref
            sPORTC |= 1<<nLED1;    //   turn on LED 1
        if (C2OUT)                 // if C2IN+ > CVref
            sPORTC |= 1<<nLED3;    //   turn on LED 3

        // test inputs 2 and 4
        CIS = 0;                   // select C1IN- and C2IN- pins as -refs
        __delay_us(10);            // wait 10us to settle
        if (C1OUT)                      // if C1IN- > CVref
            sPORTC |= 1<<nLED2;         //   turn on LED 2
        if (C2OUT)                      // if C1IN- > CVref
            sPORTC |= 1<<nLED4;         //   turn on LED 4

        // display test results
        PORTC = sPORTC;                 // copy shadow to PORTC
    }
}
```

Comparisons

Here the resource usage summary for this example:

Comp_4input

Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	48	39	1
HI-TECH PICC-Lite	32	42	4
HI-TECH C PRO Lite	31	57	4

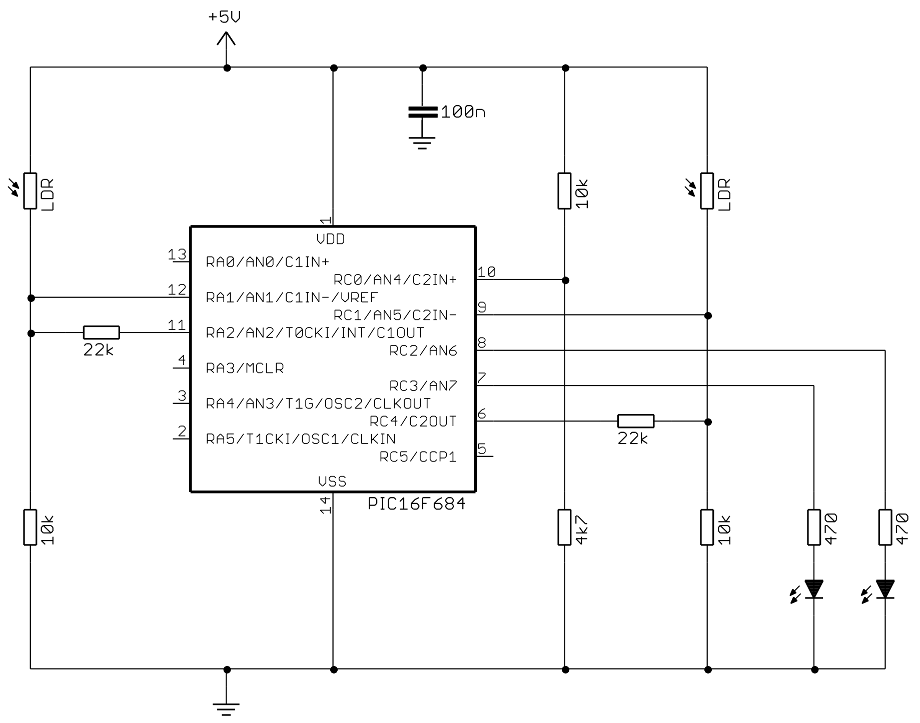
Once again, we see that the PICC-Lite compiler is able to generate highly optimised code, which is barely any larger than the hand-written assembly version. But remember that PICC-Lite can only use half of the 16F684's program memory.

Using interrupts with two comparators

We saw above that the 12F629's single comparator can be selected as an interrupt source.

As described in detail in midrange lesson 11, each of the 16F684's comparators can be selected independently as an interrupt source.

Briefly, there are now two interrupt enable bits, C1IE and C2IE, and two interrupt flags, C1IF and C2IF. Otherwise, the comparator interrupts are used in the same way as on the 12F629.



To illustrate this, we'll use the circuit on the right, which once again incorporates hysteresis, to minimise the number of interrupts triggered as the comparator inputs cross the threshold level. This is done by feeding the comparator's outputs, available on the C1OUT and C2OUT pins, back into the inputs (C1IN- and C2IN-) via 22 kΩ resistors.

The external outputs are only available in comparator mode 6 (110), and since the internal voltage reference is not available as in input in that mode, we have to use a external voltage divider, formed by the 4.7 kΩ and 10 kΩ resistors connected to C2IN+, which creates a 1.6 V reference.

As we did in the single-comparator interrupt example, we'll use the interrupt service routine to turn on an LED when its associated comparator input is higher than the 1.6 V reference (corresponding to bright light).

HI-TECH C PRO or PICC-Lite

As usual, we first need to setup the I/O port (PORTC, in this case), so that the comparator input pins on PORTC (C2IN+ and C2IN-) are configured as inputs, and the other pins as outputs:

```
    // configure ports
    TRISC = 0b000011;           // configure comparator input pins on PORTC
                                //   (RC0/C2IN+ and RC1/C2IN-) as inputs
```

There is no need to explicitly configure the comparator input pins on PORTA (C1IN- is used in this example) as inputs, as all pins are inputs by default.

To select comparator mode 6, with the outputs inverted (to create the positive feedback necessary for hysteresis), we have:

```
    // configure comparators
    CMCON0 = 0b110;             // select mode 6 (CM = 110):
                                //   C1 -ref is C1IN-,
                                //   C1 +ref is C2IN+
                                //   C2 -ref is C2IN-,
                                //   C2 +ref is C2IN+,
                                //   both external outputs enabled,
                                //   both comparators on
    C1INV = 1;                  // C1 output inverted
                                //  -> C1OUT = 1 if C1IN- > C2IN+
    C2INV = 1;                  // C2 output inverted
                                //  -> C2OUT = 1 if C2IN- > C2IN+
```

To enable the external comparator outputs, we must clear the corresponding TRIS bits:

```
                                // enable comparator external outputs:
    TRISA2 = 0;                 //   C1OUT (RA2)
    TRISC4 = 0;                 //   C2OUT (RC4)
```

The interrupts are enabled in the same way as before, except that we now have two comparators:

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

                                // enable comparator interrupts:
    CMCON0;                     //   read CMCON0 to clear mismatch
    C1IF = 0;                   //   clear interrupt flags
    C2IF = 0;
    C1IE = 1;                   //   set enable bits
    C2IE = 1;
```

The interrupt service routine is much the same as before, but with one important difference – we now have to respond to two possible input sources, so we have to check each comparator's interrupt flag, to ensure that we're servicing the correct interrupt source:

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

    if (C1IF)
    {
        // Comparator 1 interrupt
        //   Triggered on any comparator 1 output change,
        //   caused by C1IN- input crossing C2IN+ (1.6 V) threshold
        //
        CMCON0;                     // read CMCON0 to clear mismatch condition
        C1IF = 0;                   // clear interrupt flag

        // turn on LED 1 if C1IN- < C2IN+
        if (C1OUT)                  // if comparator output high (C1IN- > C2IN+)
        {
            sPORTC &= ~(1<<nLED1);  //   turn off (shadow) LED 1
        }
        else                        // else (C1IN- < C2IN+)
        {
            sPORTC |= 1<<nLED1;     //   turn on (shadow) LED 1
        }
    }

    if (C2IF)
    {
        // Comparator 2 interrupt
        //   Triggered on any comparator 2 output change,
        //   caused by C2IN- input crossing C2IN+ (1.6 V) threshold
        //
        CMCON0;                     // read CMCON0 to clear mismatch condition
        C2IF = 0;                   // clear interrupt flag

        // turn on LED 2 if C2IN- < C2IN+
        if (C2OUT)                  // if comparator output high (C2IN- > C2IN+)
        {
            sPORTC &= ~(1<<nLED2);  //   turn off (shadow) LED 2
        }
        else                        // else (C2IN- < C2IN+)
        {
            sPORTC |= 1<<nLED2;     //   turn on (shadow) LED 2
        }
    }
}
```

The main loop still does nothing but continually copy the contents of a shadow register (which is updated by the interrupt service routine) to the output pins, but this time we're using PORTC instead of GPIO:

```
    /*** Main loop ***/
    for (;;)
    {
        // continually copy shadow to port
        PORTC = sPORTC;
    }
```

Given that the program is otherwise very similar to the single-comparator interrupt example presented earlier, there is no need to present the full listing here. The source code is of course all available at www.gooligum.com.au.

Comparisons

Here again is the resource usage summary for this example:

2xComp_Interrupt

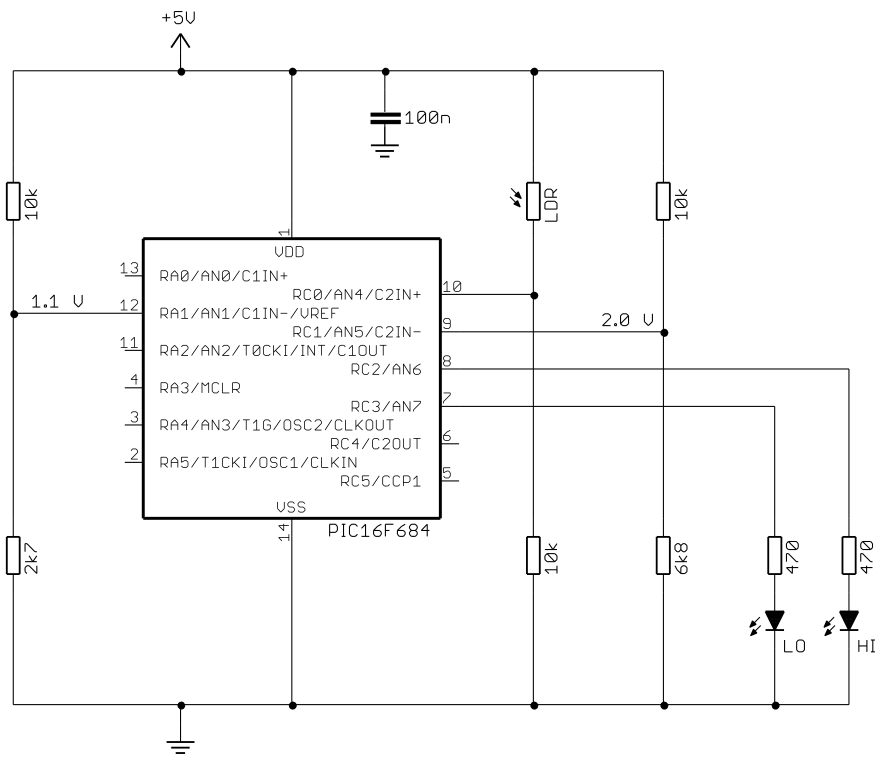
Assembler / Compiler	Source code (lines)	Program memory (words)	Data memory (bytes)
Microchip MPASM	74	59	3
HI-TECH PICC-Lite	36	57	5
HI-TECH C PRO Lite	36	96	7

The pattern is the same as for the single-comparator interrupt example, with the C source code being less than half as long as the assembler source – and in this example the PICC-Lite compiler is able to generate optimised code even smaller than the hand-written assembly version.

Wake-up when signal goes outside limits

Given that each of the 16F684's two comparators can be used as an interrupt source, it follows that each comparator can be used to wake the PIC from sleep mode.

This can be useful when you wish to monitor two inputs, and wake the PIC when either crosses a threshold. Or, you could use the two comparators to monitor a single input, comparing it against two thresholds (one for each comparator), waking the PIC when either threshold is crossed. These thresholds could be the upper and lower limits of an allowed range; the PIC waking and indicating an "alarm" condition, whenever the input goes outside these limits.



The circuit on the right was used in midrange lesson 11 to demonstrate wake-up when an input crosses either of two limits.

The most appropriate comparator mode for this application is 3 (011), where the common input is on C2IN+ and the two thresholds are defined by the voltages on C1IN- and C2IN-. In this example, a voltage divider is used to generate the low threshold of 1.1 V on C1IN-, and another voltage divider generates the high threshold of 2.0 V on C2IN-.

The LED on RC2 is used to indicate the "not enough light" condition, and the LED on RC3 indicates "too much light".

When either of the thresholds is crossed (in either direction), the PIC should wake, and if the input is outside the allowed range, the appropriate LED should be turned on, until the input is back within the allowed range. The PIC can then go back into sleep mode, to wait for the next threshold crossing.

HI-TECH C PRO or PICC-Lite

The comparator mode is selected as usual, with the inversion bits configured so that each output is a '1' only when an error condition (input too high or low) exists:

```
    // configure comparators
    CMCON0 = 0b011;             // select mode 3 (CM = 011):
                                //   C1 -ref is C1IN-,
                                //   C1 +ref is C2IN+,
                                //   C2 -ref is C2IN-,
                                //   C2 +ref is C2IN+,
                                //   no external outputs,
                                //   both comparators on
    C1INV = 1;                  // C1 output inverted
                                //  -> C1OUT = 1 if C2IN+ < C1IN-
    C2INV = 0;                  // C2 output not inverted
                                //  -> C2OUT = 1 if C2IN+ > C2IN-
```

Wake-up on comparator change is enabled in the same way as in the single-comparator example, except that there are now two comparators:

```
    // enable comparator interrupts (for wake on change)
    PEIE = 1;                   // enable peripheral interrupts
    C1IE = 1;                   // and comparator interrupts
    C2IE = 1;
```

In the main loop, we need to test each comparator output, and light the corresponding LED as long as that comparator output remains high.

The C code logically equivalent to the assembler code presented in midrange lesson 11 is:

```
        // test for and wait while input low
        while (C1OUT)           // while C2IN+ < C1IN- (1.1 V)
            LO = 1;             //   turn on "low" LED

        // test for and wait while input high
        while (C2OUT)           // while C2IN+ > C2IN- (2.0 V)
            HI = 1;             //   turn on "high" LED
```

Very short and simple!

In fact, it is a little too short; the assembler version included '`banksel`' directives which are not strictly necessary, because the registers involved in this code fragment (CMCON0 and PORTC) are in the same bank. Neither HI-TECH C compiler (not even HI-TECH C PRO, when running in the un-optimised "lite" mode) generates unnecessary bank selection code. This means that the generated C code is shorter, and faster, than the hand-written assembler version – in this case, a little too fast!

It turns out that the sudden additional current drain caused by turning on either LED connected to PORTC creates enough disruption to momentarily affect the comparator inputs. To generate consistent, correct results when reading the comparator outputs, it is necessary to add a short delay between lighting the LED and reading the comparator input.

The following PICC-Lite code works correctly:

```
        // test for and wait while input low
        while (C1OUT)            // while C2IN+ < C1IN- (1.1 V)
        {
            LO = 1;              //   turn on "low" LED
            DelayUs(2);          //   short delay to settle
        }

        // test for and wait while input high
        while (C2OUT)            // while C2IN+ > C2IN- (2.0 V)
        {
            HI = 1;              //   turn on "high" LED
            DelayUs(2);          //   short delay to settle
        }
```

For HI-TECH C PRO, with its built-in delay functions, we have instead:

```
        // test for and wait while input low
        while (C1OUT)            // while C2IN+ < C1IN- (1.1 V)
        {
            LO = 1;              //   turn on "low" LED
            __delay_us(2);       //   short delay to settle
        }

        // test for and wait while input high
        while (C2OUT)            // while C2IN+ > C2IN- (2.0 V)
        {
            HI = 1;              //   turn on "high" LED
            __delay_us(2);       //   short delay to settle
        }
```

When these loops have completed, we know that the input must be no longer too high or too low[7], so we can turn off any LEDs that had been lit:

```
        // turn off LEDs
        PORTC = 0;
```

Then we can enter sleep mode, after clearing any existing comparator mismatch condition (by reading CMCON0) and clearing the comparator interrupt flags:

```
        // enter sleep mode
        CMCON0;                  // read CMCON0 to clear comparator mismatch
        C1IF = 0;                // clear comparator interrupt flags
        C2IF = 0;
        SLEEP();
```

And when the device wakes from sleep mode, we can start the main loop again, to detect which threshold was crossed.

[7] unless it is changing very quickly...

Complete program

Here is the HI-TECH C PRO version of this example, showing how these fragments fit together:

```
*************************************************************************
*   Description:    Lesson 6, example 10                                *
*                                                                       *
*   Demonstrates wake-up on dual comparator change                      *
*                                                                       *
*   Device wakes when input crosses a low or high threshold             *
*                                                                       *
*   Turns on Low  LED when C2IN+ < C1IN- (1.1 V)                        *
*         or High LED when C2IN+ > C2IN- (2.0 V)                        *
*   then sleeps until the next change                                   *
*                                                                       *
*   (thresholds are set by external voltage dividers)                   *
*                                                                       *
*************************************************************************
*   Pin assignments:                                                    *
*       C2IN+ - voltage to be measured (e.g. pot output or LDR)         *
*       C1IN- - low threshold (1.1 V)                                   *
*       C2IN- - high threshold (2.0 V)                                  *
*       RC3   - "Low" lED                                               *
*       RC2   - "High" LED                                              *
*                                                                       *
************************************************************************/

#include <htc.h>

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


/***** 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 LO      RC3         // "Low" LED
#define nLO     3           //   (bit 3)
#define HI      RC2         // "High" LED
#define nHI     2           //   (bit 2)


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

    // configure ports
    TRISC = ~(1<<nLO|1<<nHI);   // configure LED pins (only) as outputs

    // configure comparators
    CMCON0 = 0b011;             // select mode 3 (CM = 011):
                                //   C1 -ref is C1IN-,
                                //   C1 +ref is C2IN+,
                                //   C2 -ref is C2IN-,
                                //   C2 +ref is C2IN+,
                                //   no external outputs,
                                //   both comparators on
    C1INV = 1;                  // C1 output inverted
                                //  -> C1OUT = 1 if C2IN+ < C1IN-
    C2INV = 0;                  // C2 output not inverted
                                //  -> C2OUT = 1 if C2IN+ > C2IN-

    // enable comparator interrupts (for wake on change)
    PEIE = 1;                   // enable peripheral interrupts
    C1IE = 1;                   // and comparator interrupts
    C2IE = 1;


    // Main loop
    for (;;)
    {
        // test for and wait while input low
        while (C1OUT)           // while C2IN+ < C1IN- (1.1 V)
        {
            LO = 1;             //   turn on "low" LED
            __delay_us(2);      //   short delay to settle
        }

        // test for and wait while input high
        while (C2OUT)           // while C2IN+ > C2IN- (2.0 V)
        {
            HI = 1;             //   turn on "high" LED
            __delay_us(2);      //   short delay to settle
        }

        // turn off LEDs
        PORTC = 0;

        // enter sleep mode
        CMCON0;                 // read CMCON0 to clear comparator mismatch
        C1IF = 0;               // clear comparator interrupt flags
        C2IF = 0;
        SLEEP();

        // repeat forever
    }
}
```

Comparisons

Here the resource usage summary for this example:

2xComp_Wakeup

Assembler / Compiler	**Source code (lines)**	**Program memory (words)**	**Data memory (bytes)**
Microchip MPASM	43	36	0
HI-TECH PICC-Lite	28	37	3
HI-TECH C PRO Lite	27	41	2

The optimised code generated by the PICC-Lite compiler would be smaller than the hand-written assembly version, except for the delays that had to be added to the C versions of this example, to make them work correctly.

Summary

We have seen that it is possible to effectively utilise the comparators and voltage references available on midrange PICs, such as the 12F629 and 16F684, using either of the HI-TECH C compilers.

All of the examples could be expressed clearly and succinctly in C, with the source code always being significantly shorter than the assembler equivalent:

Source code (lines)

Assembler / Compiler	**Ex 1b**	**Ex 4**	**Ex 5**	**Ex 7**	**Ex 8**	**Ex 9**	**Ex 10**
Microchip MPASM	27	60	38	45	48	74	43
HI-TECH PICC-Lite	10	28	24	25	32	36	28
HI-TECH C PRO Lite	10	28	24	24	31	36	27

In these examples, the optimised code generated by PICC-Lite was often hardly any longer, and in one case shorter, than the hand-written assembler code:

Program memory (words)

Assembler / Compiler	**Ex 1b**	**Ex 4**	**Ex 5**	**Ex 7**	**Ex 8**	**Ex 9**	**Ex 10**
Microchip MPASM	20	46	43	38	39	59	36
HI-TECH PICC-Lite	20	51	57	39	42	57	37
HI-TECH C PRO Lite	23	79	66	50	57	96	41

Data memory (bytes)

Assembler / Compiler	**Ex 1b**	**Ex 4**	**Ex 5**	**Ex 7**	**Ex 8**	**Ex 9**	**Ex 10**
Microchip MPASM	1	3	3	1	1	3	0
HI-TECH PICC-Lite	2	5	6	4	4	5	3
HI-TECH C PRO Lite	2	7	5	4	4	7	2

This suggests that there is no real penalty, for these examples, in using C – especially if you use a paid-for C compiler with full optimisation and no restrictions. When you pay for a good C compiler, you're buying the ability to write clear code, quickly (as shown by the source code length comparison), without paying a significant penalty in code size (as shown by the program memory usage comparison).

But as the last example showed, there can be a downside, in that you may not be aware of exactly what the code generated by your C compiler is doing – we had to add short delays into the polling loops, because the C compilers optimised the code a little "too well"! This isn't a bug; the compilers are doing what they should, but it shows that sometimes what's happening in detail, at a low level normally hidden by the C compiler, can be important. In assembler, you have no choice but to be across all that low-level detail...

The next lesson will focus on driving 7-segment displays (revisiting the material from midrange lesson 12), showing how lookup tables and multiplexing can be implemented using HI-TECH C.