

Introduction to PIC Programming

Midrange Architecture and Assembly Language

by David Meiklejohn, Gooligum Electronics

Lesson 10: Enhanced features of the PIC16F684

The [last lesson](#) introduced the PIC12F629's single analog comparator module – typical of the comparator modules available in older midrange PIC devices. It showed that even a single comparator can be very flexible, especially when used with a programmable voltage reference.

But sometimes (you may for example need to generate an interrupt whenever an analog signal is outside an allowed range) you really do need two comparators. Or your application may simply need more I/O pins, or memory, than the PIC12F629 can provide. That means moving up to a larger PIC!

This lesson takes a modest step forward in the 8-bit PIC landscape by introducing the 14-pin PIC16F684.

In addition to a dual comparator module (covered in the [next lesson](#)), analog-to-digital converter (see [lesson 13](#)), and other significant features such as a capture/compare/PWM (pulse width modulation) module which will be described in later lessons, the 16F684 provides a few additional features which don't really warrant a full lesson, so we'll describe them (briefly) here.

In summary, this lesson covers:

- Introduction to the PIC16F684 and its I/O ports
- The 16F684's internal oscillators
- Enhanced brown-out reset
- Enhanced watchdog timer
- Ultra Low-Power Wake-Up

Introducing the PIC16F684

The 16F684 is a 14-pin midrange PIC (giving it eleven I/O pins and one input-only pin; twice as many in total as the 12F629), with two comparators (see the [next lesson](#)) and a single 10-bit analog-to-digital converter (described in [lesson 13](#)), with eight input channels.

It also has twice as much program, data and EEPROM memory as the 12F629, and includes peripherals and features not found in the 12F629, such as a second 8-bit timer, and a capture/compare/PWM (pulse width modulation) module.

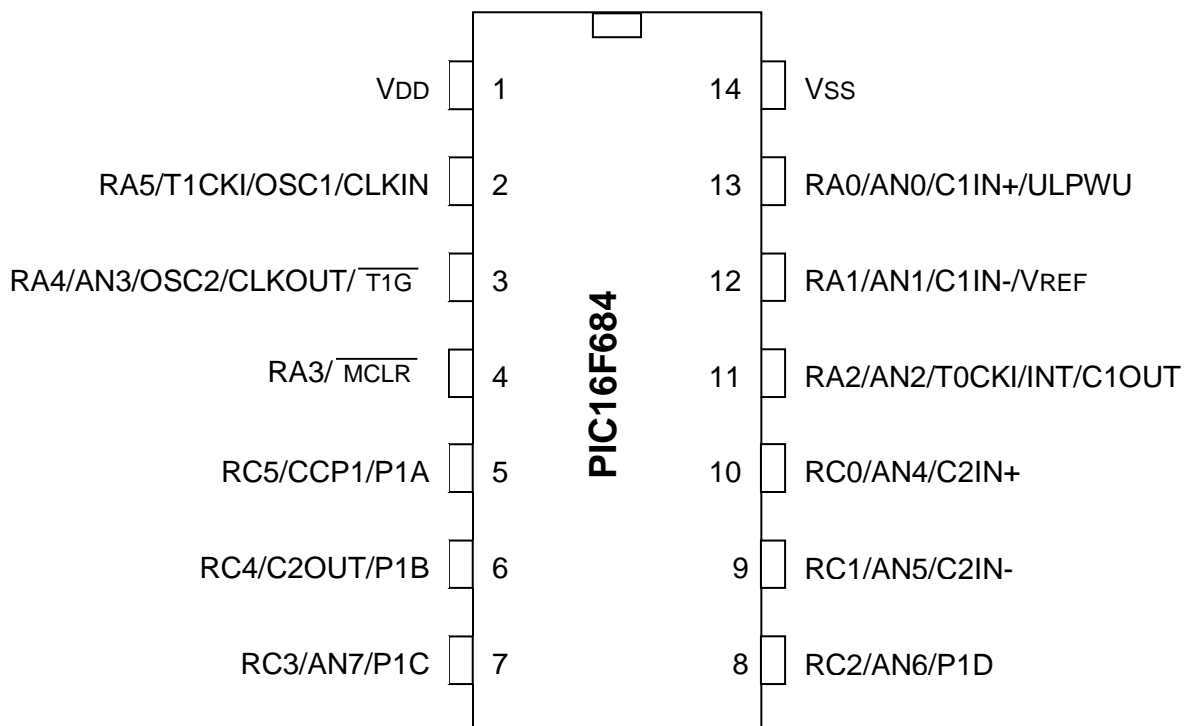
It also includes a much more flexible internal clock module, capable of generating a wide range of clock frequencies and able to take over automatically if the external clock fails. And brown-out reset is no longer an all-or-nothing proposition; it can be disabled automatically during sleep, and can be placed under software control. These features are described in this lesson.

And, of course, the 16F684 has all the advantages intrinsic to the midrange architecture, such as support for interrupts, compared to the 14-pin 16F505 and 16F506 PICs introduced in the [baseline lessons](#).

The basic features of various 8- and 14-pin PICs are summarised in the following table:

Device	I/O pins	Memory (words or bytes)			Timers		Analog		Clock rate (max MHz)
		Program	Data	EEPROM	8-bit	16-bit	Comp-arators	ADC inputs	
12F508	6	512	25	0	1	0	0	0	4
12F510	6	1024	38	0	1	0	1	3	8
12F609	6	1024	64	0	1	1	1	0	20
12F629	6	1024	64	128	1	1	1	0	20
12F683	6	2048	128	256	2	1	1	4	20
16F505	12	1024	72	0	1	0	0	0	20
16F506	12	1024	67	0	1	0	2	3	20
16F610	12	1024	64	0	1	1	2	0	20
16F684	12	2048	128	256	2	1	2	8	20
16F688	12	4096	256	256	1	1	2	8	20

It is available in a 14-pin package, as shown:



Notice that, as we move to more complex devices, most pins tend to have more functions...

The data memory, or register file, on the 16F684 is arranged in two banks, as follows:

PIC16F684 Registers

Address	Bank 0	Address	Bank 1
00h	INDF	80h	INDF
01h	TMR0	81h	OPTION_REG
02h	PCL	82h	PCL
03h	STATUS	83h	STATUS
04h	FSR	84h	FSR
05h	PORTA	85h	TRISA
06h		86h	
07h	PORTC	87h	TRISC
08h		88h	
09h		89h	
0Ah	PCLATH	8Ah	PCLATH
0Bh	INTCON	8Bh	INTCON
0Ch	PIR1	8Ch	PIE1
0Dh		8Dh	
0Eh	TMR1L	8Eh	PCON
0Fh	TMR1H	8Fh	OSCCON
10h	T1CON	90h	OSCTUNE
11h	TMR2	91h	ANSEL
12h	T2CON	92h	PR2
13h	CCPR1L	93h	
14h	CCPR1H	94h	
15h	CCP1CON	95h	WPUA
16h	PWM1CON	96h	IOCA
17h	ECCPAS	97h	
18h	WDTCON	97h	
19h	CMCON0	98h	VRCON
1Ah	CMCON1	99h	EEDAT
1Bh		9Bh	EEADR
1Dh		9Ch	EECON1
1Eh	ADRESH	9Dh	EECON2
1Fh	ADCON0	9Eh	ADRESL
20h	General Purpose Registers	9Fh	ADCON1
		A0h	General Purpose Registers
		B0h	
		C0h	
		EFh	
		F0h	Map to Bank 0 70h – 7Fh
7Fh		FFh	

The special function registers (SFRs) are laid out in much the same way as in the 12F629, but, since the 16F684 has more peripherals, it has more SFRs and therefore less unused space in the register map.

Where the same function exists for the 16F684 and 12F629, the corresponding SFRs are at the same address as in the 12F629, although in some cases with a different name. For example, 'PORTA' instead of 'GPIO', both at address 05h, and 'WPUA' instead of 'WPU', both at address 95h.

The INDF, PCL, STATUS, FSR, PCLATH and INTCON registers continue to be mapped to the same location in both banks.

All of the other SFRs appear in only one of the two banks, so it is very important to continue to use the `bankselect` directive when accessing them.

Note that the layout of the general purpose registers (GPRs) is quite different to that on the 12F629, where all of the GPRs were mapped into both banks; all data memory on the 12F629 was shared.

In contrast, most (80) of the GPRs on the 16F684 appear only in bank 0, with a further 32 GPRs appearing only in bank 1, giving a total of 112 non-shared (*banked*) data registers.

A further 16 GPRs at the top of bank 0 can also be accessed from bank 1, giving the 16F684 16 shared data registers.

This means that shared registers are a scarcer resource than in the 12F629, so you should be careful to allocate them only as necessary, such as for context saving during interrupts, or for commonly-accessed variables such as shadow registers.

In general, you should allocate most variables using ‘UDATA’ instead of ‘UDATA_SHR’, and remember to use `banksel` when accessing the banked variables.

Some of the expanded capabilities of the 16F684 are described in the following sections.

Additional I/O pins

As mentioned, the 16F684 provides twelve I/O pins (one being input-only), compared with the six available on the 12F629.

Twelve is too many pins to represent in a single 8-bit register, so instead of a single port named GPIO, the 16F684 has two ports, named PORTA and PORTC.¹

Six I/O pins are allocated to each port:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PORTA			RA5	RA4	RA3	RA2	RA1	RA0
PORTC			RC5	RC4	RC3	RC2	RC1	RC0

The direction of each I/O pin is controlled by corresponding TRIS registers:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TRISA			TRISA5	TRISA4		TRISA2	TRISA1	TRISA0
TRISC			TRISC5	TRISC4	TRISC3	TRISC2	TRISC1	TRISC0

RA3 is input only and, like GP3 on the 12F629, it shares a pin with $\overline{\text{MCLR}}$; the pin assignment being controlled by the MCLRE bit in the configuration word, as usual.

Like the 12F629, weak pull-ups are available on the 16F684 – but only on PORTA (except for RA3, where the weak pull-up on that pin is only enabled when it is configured as $\overline{\text{MCLR}}$.)

The weak pull-ups on PORTA are individually controlled by the WPUA register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WPUA	-	-	WPUA5	WPUA4	-	WPUA2	WPUA1	WPUA0

To globally enable the PORTA weak pull-ups, clear the $\overline{\text{RAPU}}$ bit in the OPTION register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OPTION_REG	$\overline{\text{RAPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0

¹ Why ‘PORTC’ instead of ‘PORTB’? “Historical reasons...”

This is analogous to the WPU register and $\overline{\text{GPPU}}$ bit in the 12F629, described in [lesson 3](#).

Similarly, the 16F684 supports interrupt-on-change (see [lesson 7](#)), but only for the pins in PORTA.

Each pin in PORTA can be individually configured for interrupt-on-change by setting the corresponding bit in the IOCA register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IOCA	-	-	IOCA5	IOCA4	IOCA3	IOCA2	IOCA1	IOCA0

A mismatch condition on any pin configured for interrupt-on-change is indicated by the RAIF flag, in the INTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
INTCON	GIE	PEIE	T0IE	INTE	RAIE	T0IF	INTF	RAIF

PORTA change interrupts are enabled by setting the RAIE bit.

These features correspond to the 12F629's IOC register and GPIE and GPIF bits; only the names are different.

As mentioned in the [last lesson](#), any pins with an analog function, whether comparator or ADC inputs start in analog mode, with the digital input buffer disconnected; the digital input will read as '0'.

On the 16F684, pins RA0, RA1, RA2, RA4, RC0, RC1, RC2 and RC3 can be selected as ADC inputs, and therefore are configured as analog inputs by default. To use any of these pins as digital inputs, they must first be deselected as analog inputs, as explained in [lesson 13](#) on analog-to-digital conversion.

For now, it is enough to know that clearing the ANSEL register will deselect all of the ADC inputs, enabling all of these pins as digital inputs – as long as any comparator inputs on those pins are also disconnected. We'll see how to do that, later in this lesson.

Additional clock options

The 16F684 supports the same external clock sources as the 12629, including EC, LP, XT, HS and RC modes, as described in [lesson 8](#).

Additionally, the 16F684 includes a much more flexible internal clock module, able to generate a range of frequencies. It is also possible to switch between the internal and external clock sources under software control, "on the fly". And the internal clock can be configured to take over automatically, should the external clock fail, and can be used temporarily while the external clock stabilises, minimising delays and saving power in low-power applications.

The internal oscillator is controlled by the OSCCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
OSCCON	-	IRCF2	IRCF1	IRCF0	OSTS	HTS	LTS	SCS

The IRCF<2:0> bits are used to select the internal oscillator frequency, as follows:

IRCF<2:0>	Oscillator	Frequency
000	LF	31 kHz (approx)
001	HF	125 kHz
010	HF	250 kHz
011	HF	500 kHz
100	HF	1 MHz
101	HF	2 MHz
110	HF	4 MHz (default)
111	HF	8 MHz

There are actually two independent internal RC oscillators in the 16F684: an uncalibrated low frequency oscillator, 'LFINTOSC', running at approximately 31 kHz, and a high frequency oscillator, 'HFINTOSC', factory-calibrated to run at 8 MHz \pm 1% (at 25°C).

The low frequency oscillator is selected as the internal clock source when IRCF<2:0> = 000. Since this oscillator is not calibrated, you should only select this mode when the exact clock rate is unimportant, compared with saving power (lower clock rates mean lower power consumption).

The 8 MHz oscillator is used as the clock source in the remaining modes, divided by a postscaler to generate frequencies from 125 kHz to 8 MHz. The default frequency is 4 MHz, for compatibility with other PICs. If you want your code to run faster, you can select 8 MHz, or if you don't need so much speed and wish to save power, you can select a lower speed.

The factory calibration value in the 16F684² is automatically used to set the frequency of the HFINTOSC; unlike the 12F629, there is no need for you code to load a calibration value.

However, if you need to adjust the speed of the HFINTOSC, you can do so by writing a non-zero value to the OSCTUNE register; see the 16F684 data sheet for details.

The internal clock source (LFINTOSC or HFINTOSC) is selected whenever the SCS bit is set, regardless of the setting in the processor configuration word.

Otherwise, if SCS is clear (which it is, by default), the clock source is selected by the FOSC<2:0> bits in the configuration word, in the same way as for the 12F629:

13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
-	-	FCMEN	IESO	BOREN1	BOREN0	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	MCLRE	$\overline{\text{PWRTE}}$	WDTE	FOSC2	FOSC1	FOSC0

The FCMEN configuration bit is used to enable the "Fail-Safe Clock Monitor (FSCM)", which monitors the external clock (assuming you have selected an external clock mode, of course!) and, if it detects that the external clock has failed, automatically switches to the internal clock source (the speed of which is set by IRCF<2:0>, as usual), and sets an interrupt flag (OSFIF in the PIR1 register) to indicate that this the external oscillator has failed, so that your program can take appropriate action, and generates an interrupt if the OSFIE bit in the PIE1 register is set.

This is useful if you are using an external oscillator, such as a crystal, for better timing accuracy – your code can keep running, even if the external oscillator fails.

² The factory calibration value is stored in a memory location, the "Calibration Word", which cannot be accessed by your program code. It is not erased if the PIC is reprogrammed correctly.

The IESO configuration bit enables “Two-Speed Start-up” mode.

As explained in [lesson 8](#), it can take a while for a crystal oscillator to commence stable oscillation, so an oscillator start-up timer (OST) is used to delay program execution while the external crystal oscillator stabilises.

Two-Speed Start-up mode allows this delay to be bypassed, by initially using the internal oscillator for program execution, until the external oscillator is stable. Once the external oscillator is stable, it is selected as the clock source.

This is especially useful in low-power applications which use an external oscillator. Since the external oscillator is disabled in sleep mode, the OST delay applies when the PIC wakes from sleep mode, and this delay wastes precious power. Two-Speed Start-up allows the PIC to wake and execute code immediately, using the internal clock, and then return to sleep mode more quickly (saving power), even in applications where an external oscillator is normally used.

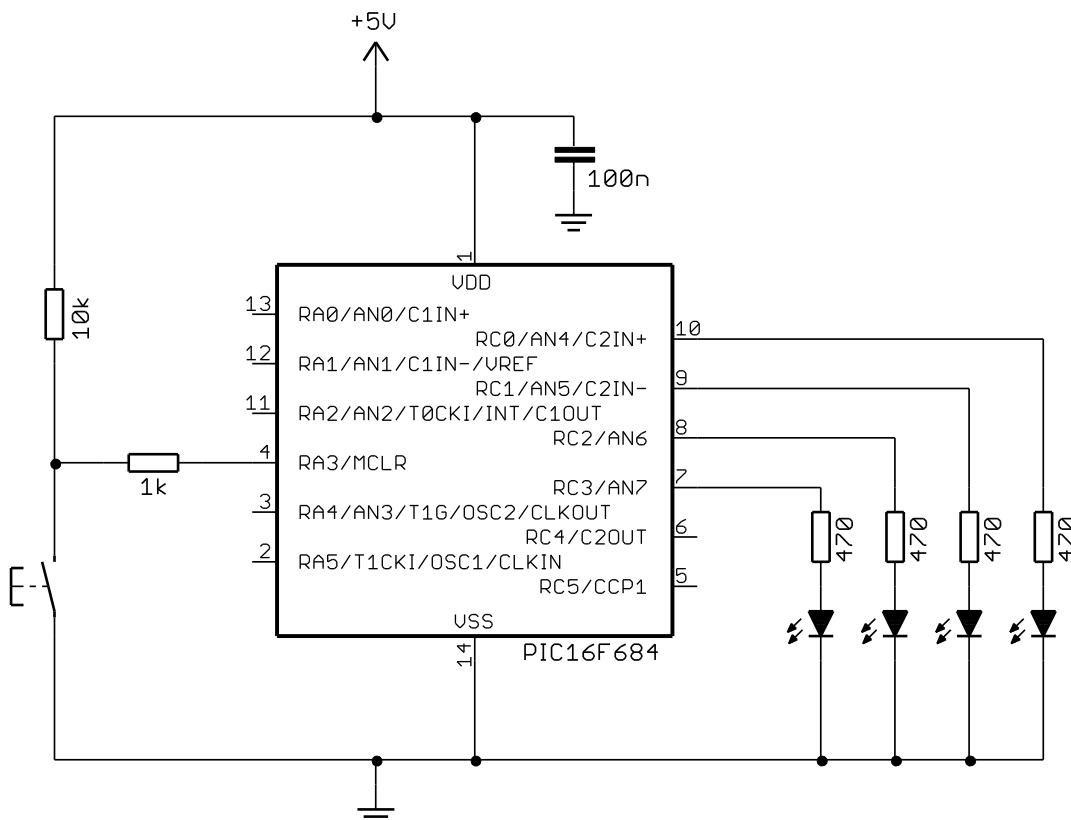
To check whether the PIC is using an internal or external clock source, your code can test the OSTS bit on the OSCCON register; it is cleared when the internal clock is being used.

Your code can switch to the internal clock at any time, but before doing so, you may wish to check that it is stable. You can do this by checking the HTS and LTS bits in the OSCCON register, which indicate that the high-frequency and low-frequency oscillators are stable, respectively.

Example 1: Changing the internal clock frequency

We can demonstrate how to change the internal clock frequency on the fly, with a simple example:

The Low Pin Count Demo Board we’ve been using in these lessons has four LEDs connected to RC0 – RC3, as shown below. Note that there is no need, when using a 14-pin PIC, to jumper any connections on the 14-pin header, as there was with 8-pin devices such as the PIC12F629.



We can flash these LEDs at varying rates, by varying the processor clock frequency.

Since the internal clock can generate a wide range of frequencies, from 31 kHz to 8 MHz (a range of 1:256), it is difficult to represent that range with a single LED.

Suppose we adapted our code from [lesson 1](#), using a 4 MHz processor clock to flash a single LED at 1 Hz. If we took this same delay code, which assumes a 4 MHz clock, and clocked the processor at 8 MHz instead, the LED would flash twice as fast, at 2 Hz (twice per second) instead of 1 Hz.

That's ok – a 2 Hz flash rate is easy enough to see. But suppose we drop the processor clock to 31 kHz. The same delay code, designed to generate a 500 ms delay from a 4 MHz clock, would take 128 times longer to run at 31 kHz, generating a 64 second delay, and flashing the LED once every 128 seconds! That's a long time to wait to see it change. But if we shorten the delay, so that the LED flashes at a reasonable rate (say 0.25 Hz) given a 31 kHz clock, it would be flashing at 64 Hz when the processor clock is increased to 8 MHz, making it impossible to see the flashing!

To get around this problem, we can make the four LEDs flash at different rates – and an effective way to do that is to implement a binary count. Simply increment `PORTC`³ every 125 ms, and the LED on `RC0` will flash at 4 Hz, `RC1` at 2 Hz, `RC2` at 1 Hz, and `RC3` at 0.5 Hz.

If we set it up so that `RC3` flashes at 4 Hz with a 4 MHz processor clock, `RC3` will flash at 8 Hz (still visibly flashing) when the processor clock is increased to 8 MHz, and `RC0` will flash at 0.25 Hz (once every four seconds) when the clock rate is reduced to 31 kHz. This seems to be a good compromise.

To flash `RC3` at 4 Hz, we need to increment `PORTC` at 64 Hz, or every 15.6 ms.

We can generate a delay of approximately 15400 instruction cycles with the following code, adapted from [lesson 2](#):

```

        banksel dc2                ; delay 1+15399 = 15400 cycles
dly2    movlw    .20                ; repeat inner loop 20 times
        movwf   dc2                ; -> 20x(767+3)-1 = 15399 cycles
dly1    clrf    dc1                ; inner loop = 256x3-1 = 767 cycles
        decfsz  dc1,f
        goto    dly1
        decfsz  dc2,f                ; end outer loop
        goto    dly1

```

This gives a delay of 15.4 ms with a 4 MHz processor clock (corresponding to a 1 MHz instruction clock); close enough to the 15.6 ms we were looking for. The delay will be 7.7 ms with an 8 MHz processor clock, and approximately 2 s with the 31 kHz clock.

This delay code is then placed within a loop which counts from 0 to 15 and displays each value (in binary) on the LEDs.

Instead of incrementing `PORTC` directly (see footnote), we define a variable:

```

        UDATA_SHR
dsp_cnt res 1                ; display count (appears on LEDs)

```

and use this to hold the current value of the count.

³ Directly incrementing the port register will not work correctly – partly because of the potential for read-modify-write (r-m-w) problems, as we've discussed before, but mainly because any analog (ADC and/or comparator) inputs enabled on the port will read as '0', instead of the newly-incremented value. This can be avoided by disabling all of the analog inputs, but given the possibility of r-m-w problems, a better, more robust approach is to use a shadow register.

A shared register was used to hold 'dsp_cnt' because it simplifies the code (no 'banksel' directives needed) – and to demonstrate that, unlike the PIC12F629, the PIC16F684 has both banked and unbanked GPRs, so we use both types of data memory in this example.

This count is written to the LEDs at the start of the loop. It is then displayed for whatever period the delay code generates at the current clock rate, before it is incremented. Finally, we check to see if we've counted 16 times yet, and go around the loop again if not:

```
; Complete 4-bit binary count and display on LEDs
    clrf    dsp_cnt        ; start with count = 0
count
    ; display current count
    movf    dsp_cnt,w      ; copy display count to PORTC
    banksel PORTC
    movwf   PORTC

    ; delay 15400 cycles (15.4 ms with 4 MHz processor clock)
    <DELAY CODE GOES HERE>

    ; increment count until we have counted to 16
    incf    dsp_cnt,f      ; increment display count
    movf    dsp_cnt,w
    xorlw   0x10           ; if we have not yet counted to 16,
    btfss   STATUS,Z
    goto    count         ; keep counting
```

Having completed the full binary count, at the current clock rate, we can increment the IRCF field in the OSCCON register, to select the next value:

```
; Increment processor clock frequency
    banksel OSCCON
    movlw   1<<IRCF0      ; increment IRCF field
    addwf   OSCCON,f      ; to select next clock frequency
```

Note that, because the IRCF bits are in the middle of the OSCCON register, we need to add the value '1<<IRCF0' (where 'IRCF0' is defined as '4' in the processor include file), so that only the IRCF bit-field is incremented. Note also that we don't have to worry about the IRCF bits overflowing, because any overflow will be into bit 7 of OSCCON, which is unused. But if this bit wasn't unused, we'd need to use some masking operations to protect the contents of any higher bits.

After selecting the next clock frequency, we can go back and repeat the binary count, and the new clock rate.

Before the main loop, of course, we need to initialise PORTC (PORTA is not used in this example), and set the initial internal oscillator frequency:

```
; setup ports
banksel PORTC        ; start with PORTC clear
clrf    PORTC
banksel TRISC        ; configure all PORTC pins as outputs
clrf    TRISC
; configure oscillator
movlw   b'00000001'  ; configure internal oscillator:
                    ; -000----          31 kHz (IRCF = 0)
                    ; -----1          select internal clock (SCS = 1)
banksel OSCCON
movwf   OSCCON
```

Note that, when initialising OSCCON, we're explicitly selecting the internal clock, by setting the SCS bit.

This isn't actually necessary (we could in fact simply clear OSCCON here), because we have specified the internal clock in the processor configuration word:

```

; 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 & _BOD_OFF & _WDT_OFF &
            _PWRTE_ON & _INTOSCIO & _FCMEN_OFF & _IESO_OFF

```

Note the use of '`_INTOSCIO`' instead of the '`_INTRC_OSC_NOCLKOUT`' we've been using in the 12F629 examples. The '`p16F684.inc`' include file defines both symbols, so why not use the shorter one?

Finally, note that there is no need calibrate the internal RC oscillator; the calibration value is loaded automatically at power-on, as mentioned above.

Complete program

Here is how all of these pieces fit together:

```

;*****
;
; Description:    Lesson 10, example 1
;
; Demonstrates varying internal oscillator frequencies
;
; Performs binary count on four LEDs
; incrementing at rates from 0.5 Hz to 128 Hz
; corresponding to processor clock speeds from 31 kHz to 8 MHz
;
;*****
;
; Pin assignments:
;   RC0-3 - output LEDs
;
;*****

list          p=16F684
#include      <p16F684.inc>

errorlevel   -302           ; no warnings about registers not in bank 0

;***** 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 & _BOD_OFF & _WDT_OFF & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

;***** VARIABLE DEFINITIONS
          UDATA
dc1      res 1           ; delay loop counters
dc2      res 1

          UDATA_SHR
dsp_cnt  res 1           ; display count (appears on LEDs)

```

```

;*****
RESET    CODE    0x0000          ; processor reset vector

;***** Initialisation
        ; setup ports
        banksel PORTC          ; start with PORTC clear
        clrf    PORTC
        banksel TRISC         ; configure all PORTC pins as outputs
        clrf    TRISC
        ; configure oscillator
        movlw   b'00000001'    ; configure internal oscillator:
        ; -000----            31 kHz (IRCF = 0)
        ; -----1            select internal clock (SCS = 1)
        banksel OSCCON
        movwf   OSCCON

;***** Main loop
loop
; Complete 4-bit binary count and display on LEDs
        clrf    dsp_cnt        ; start with count = 0
count
        ; display current count
        movf    dsp_cnt,w      ; copy display count to PORTC
        banksel PORTC
        movwf   PORTC

        ; delay 15400 cycles (15.4 ms with 4 MHz processor clock)
        banksel dc2
dly2    movlw   .20            ; repeat inner loop 20 times
        movwf   dc2           ; -> 20x(767+3)-1 = 15399 cycles
        clrf    dc1           ; inner loop = 256x3-1 = 767 cycles
dly1    decfsz  dc1,f
        goto    dly1
        decfsz  dc2,f         ; end outer loop
        goto    dly1

        ; increment count until we have counted to 16
        incf    dsp_cnt,f     ; increment display count
        movf    dsp_cnt,w
        xorlw   0x10          ; if we have not yet counted to 16,
        btfss  STATUS,Z
        goto    count        ; keep counting

; Increment processor clock frequency
        banksel OSCCON
        movlw   1<<IRCF0     ; increment IRCF field
        addwf   OSCCON,f     ; to select next clock frequency

        goto    loop        ; repeat forever

END

```

Enhanced Brown-out Reset (BOR)

Brown-out detection, where the PIC is configured to respond to a sagging supply voltage (a brown-out) by holding the device in reset until the voltage recovers, was described for the PIC12F629 in [lesson 8](#).

The PIC16F684 implements the same functionality, but gives you more control over when brown-out detection (referred to as brown-out reset, or BOR, in the 16F684 data sheet) is enabled.

For the 12F629, brown-out detection is all-or-nothing; it is either enabled (using the BODEN bit in the configuration word), or not.

A problem with this is that brown-out detection requires some current, and may not be justifiable when you are trying to save as much power as possible – as is often the case, when in sleep mode.

To address this, the 16F684 provides a mode where BOR is enabled when the device is running, but is automatically disabled in sleep mode.

Or, you may prefer to enable and disable BOR at runtime, under software control. For example, you may wish to enable BOR when running from mains power (because it may sag temporarily), but not when running from batteries (perhaps because you have another means of monitoring battery voltage, and if battery voltage falls, it's a one-way trip so you'll want to shut the PIC down cleanly, instead of being subject to a brown-out reset).

The brown-out reset mode is controlled by the BOREN<1:0> bits in the configuration word:

13	12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
-	-	FCMEN	IESO	BOREN1	BOREN0	$\overline{\text{CPD}}$	$\overline{\text{CP}}$	MCLRRE	$\overline{\text{PWRTE}}$	WDTE	FOSC2	FOSC1	FOSC0

These bits select the BOR mode, as follows:

BOREN<1:0>	BOR mode
00	disabled
01	software control
10	enabled when device running and disabled during sleep
11	enabled

When BOREN<1:0> = '01', brown-out reset is controlled by the SBOREN bit in the PCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCON	-	-	ULPWUE	SBOREN	-	-	$\overline{\text{POR}}$	$\overline{\text{BOD}}$

BOR is enabled when SBOREN is set, and disabled when SBOREN is cleared.

Enhanced watchdog timer (WDT)

As we saw in [lesson 7](#), the PIC12F629 includes a watchdog timer (WDT), which can be used to reset the device automatically if it crashes / stops responding, or to regularly wake the device from sleep mode.

Associated with the WDT is a time-out period; if the program does not execute a 'clrwdt' instruction within this period, the device will be reset. Alternatively, if the device is in sleep mode, it will wake at the end of the WDT time-out period.

The WDT time-out on the PIC12F629 (and in the baseline PICs) is 18 ms.

This can be extended by using a prescaler with a ratio of up to 1:128, giving a time-out period of up to 2.3 s.

That's ok – a WDT time-out of between 18 ms and 2.3 s is more than adequate in most cases. A shorter period is good if you want your application to recover quickly from crashes, while a longer period is often better when you want to wake regularly from sleep mode.

But there's a problem. On the PIC12F629, and in the baseline PIC architecture, the WDT prescaler is shared with Timer0. If you're using the prescaler with Timer0, you're stuck with the base 18 ms WDT time-out period. Or, if you need a longer WDT time-out, you can't use the prescaler with Timer0.

The PIC16F684 addresses this problem by introducing a 16-bit prescaler dedicated to the watchdog timer, able to generate a prescale ratio of up to 1:65536.

The WDT prescale ratio is selected by the WDTPS bits in the WDTCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
WDTCON	–	–	–	WDTPS3	WDTPS2	WDTPS1	WDTPS0	SWDTEN

The following prescale ratios are available:

WDTPS<3:0> bit value	WDT prescale ratio	time-out period (nominal)
0000	1 : 32	1.0 ms
0001	1 : 64	2.1 ms
0010	1 : 128	4.1 ms
0011	1 : 256	8.3 ms
0100 (default)	1 : 512	17 ms
0101	1 : 1024	33 ms
0110	1 : 2048	66 ms
0111	1 : 4096	130 ms
1000	1 : 8192	260 ms
1001	1 : 16384	530 ms
1010	1 : 32768	1.1 s
1011	1 : 65536	2.1 s

The watchdog timer is driven by the 31 kHz uncalibrated low-frequency oscillator – the LFINTOSC oscillator described in the section on internal clocks, above.

As you can see, you can select a WDT prescale ratio between 1:32 and 1:65536. These ratios generate nominal time-out periods ranging from 1 ms and 2.1 s, as shown.

This is a wider range than available on the 12F629, although, for compatibility, the default time-out period is approximately 17 ms; close to the 12F629's nominal 18 ms.

If you need a longer time-out period than 2.1 s is inadequate, you can use the shared "Timer0" prescaler, in addition to this dedicated WDT prescaler, in the usual way (but if you're using the shared prescaler for the watchdog timer, you can't also use it for Timer0 – it's still one or the other).

Once we have selected the maximum WDT prescale ratio of 1:65536, to generate a WDT time-out period of (nominally) 2.1 s, we can configure the shared Timer0 prescaler to extend this by setting the PSA bit in OPTION_REG to '1', and using the PSA<2:0> bits to select the shared prescale ratio, as we have done before:

PS<2:0> bit value	shared prescale ratio	time-out period (nominal)
000	1 : 1	2.1 s
001	1 : 2	4.2 s
010	1 : 4	8.4 s
011	1 : 8	17 s
100	1 : 16	34 s
101	1 : 32	67 s
110	1 : 64	134 s
111	1 : 128	268 s

As you can see, the maximum achievable WDT time-out period is 268 seconds, although this is at the cost of not being able to use a prescaler with Timer0.

Another enhancement in the PIC16F684 is the ability to turn off the watchdog timer under software control.

Like the 12F629, the watchdog timer on the PIC16F684 is controlled by the WDTE bit in the processor configuration word. When WDTE is set to '1', the WDT is enabled unconditionally, and will run, whatever your program does.

But sometimes you might like to be able to turn it off; for example, to save power in sleep mode. Or perhaps you're using the WDT to wake the device periodically from sleep, but don't want to have it enabled during code execution, to avoid the need to periodically run 'clrwdt' instructions.

On the PIC16F684, when the WDTE processor configuration bit is cleared, the WDT will be initially disabled on power-on, but your program can enable the WDT by setting the SWDTEN bit in the WDTCON register, and disable it again by clearing SWDTEN.

Example 2: Long duration periodic wake from sleep

To demonstrate that the WDT on the 16F684 can be configured with a much longer time-out period than the WDT on the 12F629, we can adapt the periodic wake-up example from [lesson 7](#).

We can keep the circuit from in the last example, using the LED on RC0 as the "wake-up" indicator.

Instead of the indicator LED flashing then the device wakes after 2.3 seconds of sleep, we'll configure the WDT to time out (waking the PIC) after around 34 seconds – although you can of course choose different prescale ratios, to make this time longer or shorter, if you wish.

To use the watchdog timer, you would normally enable it in the processor configuration:

```

; ext reset, no code or data protect, no brownout detect,
; watchdog enabled, power-up timer, int clock with I/O,
; no failsafe clock monitor, two-speed start-up disabled
__CONFIG _MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_ON & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

```

But so that we can demonstrate software control of the watchdog timer, we'll configure the processor with the WDT disabled:

```

; 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 & _BOD_OFF & _WDT_OFF & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

```

and then software-enable the WDT, by setting SWDTEN, before entering sleep mode:

```

banksel WDTCON          ; enable watchdog timer
bsf      WDTCON,SWDTEN

sleep                               ; enter sleep mode (until WDT time-out)

```

The code is otherwise very much the same as the example in lesson 7, except of course for the watchdog timer prescaler configuration:

```

; configure watchdog timer
movlw   b'00010110'      ; configure WDT prescaler:
; ---1011-              WDT prescale = 65536 (WDTPS = 1011)
banksel WDTCON          ; -> WDT timeout = 2.1 s
movwf   WDTCON
movlw   1<<PSA | b'100' ; assign shared prescaler to WDT (PSA = 1)
; prescale = 16 (PS = 100)
banksel OPTION_REG      ; -> WDT timeout = 16 x 2.1 s
movwf   OPTION_REG      ; = 34 s

```

Note that there are two prescalers to configure now.

With both prescalers in use, the combined ratio is $65536 \times 16 = 1,048,576$, scaling the 31 kHz output of the LFINTOSC oscillator down to a time-out period of 34 seconds.

The 'DelayMS' macro developed in [lesson 5](#) is used to generate the 1 second delay needed to flash the LED. This macro is a wrapper for the 'delay10' subroutine developed in [lesson 2](#), but with a slight change, now that we're using a 16F684.

When we developed the 'delay10' routine for the 12F629, we had to allocate shared data memory for the variables, because the 12F629 only has shared data memory.

We could re-use the same code for the 16F684 (it would work without any problems), but it would be wasteful to do that, because, as mentioned in the introduction, the 16F684 has comparatively few shared registers; most of its data memory is banked.

So for the 16F684 (and most midrange PICs), it is better to allocate banked (non-shared) data memory for the variables, and then the code has to be modified, to include a 'banksel' directive at the start.

For reference, here is the "banked data memory" version of the 'delay10' external module:

```

;*****
; Description:   Variable Delay : N x 10ms (10ms - 2.55s)      *
;                                                       *
;               N passed as parameter in W reg              *
;               exact delay = W x 10.015ms                  *
;                                                       *
; Returns: W = 0                                           *
; Assumes: 4MHz clock                                       *
;                                                       *
;*****

```

```

#include    <p16F684.inc>    ; any midrange device will do

GLOBAL    delay10

;***** VARIABLE DEFINITIONS
          UDATA
dc1       res 1                ; delay loop counters
dc2       res 1
dc3       res 1

;*****
          CODE

delay10   ; delay W x 10ms
          ; -> ?+1+Wx(3+10009+3)-1+4 = Wx10.015ms
          banksel dc3
          movwf   dc3

dly2     movlw   .13           ; repeat inner loop 13 times
          movwf   dc2           ; -> 13x(767+3)-1 = 10009 cycles
          clrf    dc1           ; inner loop = 256x3-1 = 767 cycles
dly1     decfsz  dc1,f
          goto    dly1
          decfsz  dc2,f           ; end middle loop
          goto    dly1
          decfsz  dc3,f           ; end outer loop
          goto    dly2

          return

          END

```

Complete program

And here is the new, longer-duration, “periodic wake from sleep” program, including the code fragments above:

```

;*****
;
; Description:    Lesson 10, example 2
;
; Demonstrates long-duration periodic wake from sleep,
; using the watchdog timer
;
; Turn on LED for 1 s, turn off, then sleep for 34 s
; (LED flashes for 1 s every 35 s
;
;*****
;
; Pin assignments:
; RC0 - indicator LED
;
;*****

list      p=16F684
#include   <p16F684.inc>

#include   <stdmacros-mid.inc>    ; DelayMS - delay in milliseconds

```

```

errorlevel -302 ; no "register not in bank 0" warnings
errorlevel -312 ; no "page or bank selection not needed" messages

radix      dec

EXTERN     delay10          ; W x 10ms 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     _MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

; pin assignments
constant     nLED=0          ; indicator LED on RC0

;*****
RESET        CODE    0x0000          ; processor reset vector

;***** Initialisation
                ; configure port
movlw        ~(1<<nLED)          ; configure LED pin as output
banksel     TRISC
movwf       TRISC
                ; configure watchdog timer
movlw        b'00010110'        ; configure WDT prescaler:
                ; ---1011-          WDT prescale = 65536 (WDTPS = 1011)
banksel     WDTCON              ; -> WDT timeout = 2.1 s
movwf       WDTCON
movlw        1<<PSA | b'100'      ; assign shared prescaler to WDT (PSA = 1)
                ; prescale = 16 (PS = 100)
banksel     OPTION_REG          ; -> WDT timeout = 16 x 2.1 s
movwf       OPTION_REG          ; = 34 s

;***** Main loop
loop
banksel     PORTC              ; turn on LED
bsf         PORTC,nLED

DelayMS     1000              ; delay 1s

banksel     PORTC              ; turn off LED
bcf         PORTC,nLED

banksel     WDTCON              ; enable watchdog timer
bsf         WDTCON,SWDTEN

sleep                               ; enter sleep mode (until WDT time-out)

goto        loop              ; repeat forever

END

```

Ultra Low-Power Wake-up

We've seen that the watchdog timer can be used to periodically wake a PIC from sleep mode.

That's a convenient solution, requiring no external components, but the amount of current drawn by the watchdog timer is a concern.

According to the data sheet, given a 3.0 V power supply, no external current draw (all I/O pins in input mode and tied to VDD) and all power-consuming options disabled, the 16F684 typically draws only 150 nA in sleep mode.

However, the WDT, if enabled, with a 3.0 V supply, typically draws 2.0 μ A.

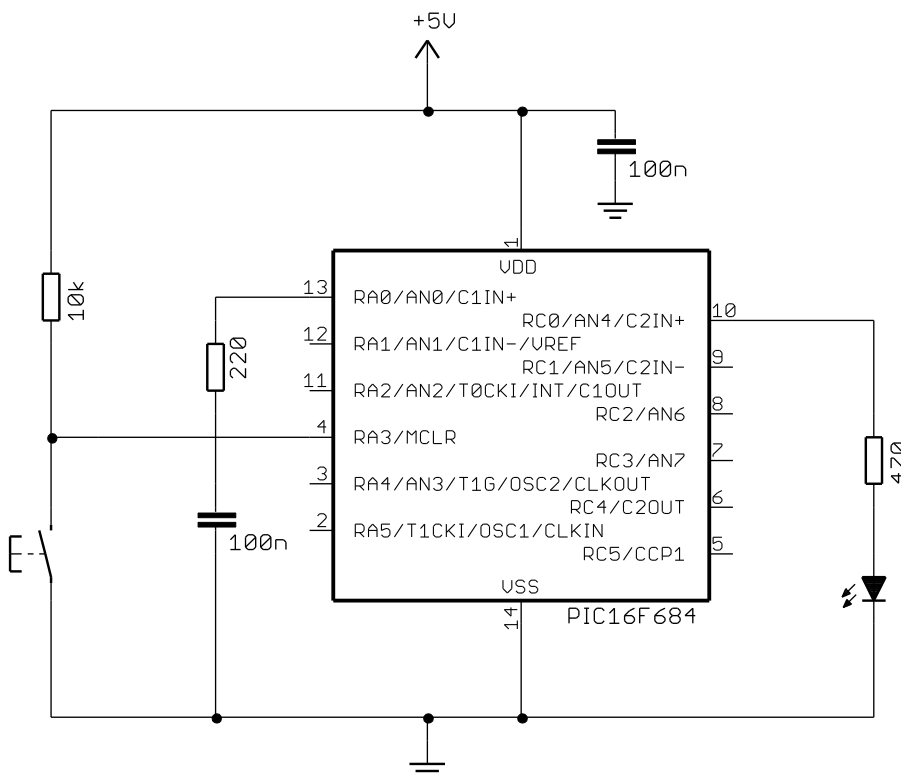
That's more than ten times as much as the device core needs in sleep mode. So, using the WDT for periodic wake-up, with a 3.0 V power supply, increases the sleep current from 0.15 μ A to 2.15 μ A – and increase of 1333%! Or, looking at it another way, your battery-powered device may last less than one tenth as long, if the WDT is enabled in sleep mode.

Another way to wake a PIC from sleep mode is to charge a capacitor, connected to one of the PIC's pins (by placing the pin in output mode, and outputting a high voltage, long enough to sufficiently charge the capacitor), and then, just before entering sleep mode, make a change that will allow the capacitor to slowly discharge. When the voltage falls below a threshold, some mechanism or other wakes the PIC.

"Voltage falls below a threshold" sounds like a comparator, and we could use a comparator change to wake the PIC (see the [last lesson](#)), but the comparator module on the 16F684 typically draws 60 μ A, with a 3.0 V supply – so clearly that's not a helpful approach for saving power!

Or, you could simply use the fact that digital inputs trigger at certain levels (a digital 'low' on a TTL input being less than 0.6 V or so), and connect the capacitor "directly" to a digital input, such as RA0, as shown in the circuit below. In practice, a series resistor should be used, to limit the charging current, as shown.

The problem with this approach is that, as mentioned before, a normal digital input will consume a high current when presented with a voltage outside the normal digital range, especially if it is slowly changing.



The PIC16F684 addresses this problem by providing a special "ultra low-power wake-up" (ULPWU) option on pin RA0.

In ULPWU mode, the RA0 pin will sink a steady (low) current, regardless of the input level.

This current, I_{ULP}, is typically only 200 nA on the 16F684, giving a total current draw of around 350 nA in sleep mode, with a 3.0 V supply.

This means that the ULPWU approach is many times more efficient than using the watchdog timer!

ULPWU mode is enabled by setting the ULPWUE bit in the PCON register:

	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PCON	-	-	ULPWUE	SBOREN	-	-	$\overline{\text{POR}}$	$\overline{\text{BOD}}$

Once the RA0 pin is in ULPWU mode, it can be used as an interrupt-on-change pin, in the same way as described for the PIC12F629's GPIO pins in [lesson 7](#).

Example 3: Ultra low-power wake-up

To demonstrate how to use the ultra low-power wake-up mode, we'll re-implement the previous periodic wake from sleep example, using ULPWU instead of the WDT, with the circuit above.

We'll turn on the LED for 1 s, as before.

During this time, we'll also charge the 100 nF capacitor on RA0.

Note the 220 Ω series resistor. As mentioned, this is used to limit the capacitor charging current. Each I/O pin on the 16F684 can supply no more than 25 mA. Assuming that when the circuit is powered on, the capacitor is initially discharged, the voltage across the resistor will be almost 5 V when RA0 outputs a 'high'. This resistor therefore limits the initial charging current to $5 \text{ V} \div 220 \Omega = 23 \text{ mA}$, which RA0 can safely supply. And this current will fall, as the capacitor charges.

As a rule of thumb, a capacitor charged through a resistor will charge to within 1% of its final value after 5 RC time constants. In this case, $5 \times 220 \Omega \times 100 \text{ nF} = 110 \mu\text{s}$, so we really only need about 1 ms to charge the capacitor. Charging it for the full second that the LED is lit is overkill, but doesn't hurt; at least we can be sure that it's fully charged!

To charge the capacitor, we simply configure RA0 as an output, and set it 'high':

```
; charge ULPWU capacitor
banksel TRISA           ; configure RA0 as an output
bcf     TRISA,0
banksel PORTA           ; set RA0 high
bsf     PORTA,0
```

After turning on the LED for one second (and then turning it off), we need to start the capacitor discharge, through RA0's ULPWU current sink.

This is done by configuring RA0 as an input, and enabling ULPWU mode:

```
; setup ULP wake-up
banksel PCON           ; enable ULPWU mode
bsf     PCON,ULPWUE
banksel TRISA           ; configure RA0 as an input
bsf     TRISA,0
```

The voltage on RA0 will now begin to steadily fall, at a rate of (IULP / C) volts/second.

With $C = 100 \text{ nF}$ and $I_{ULP} = 200 \text{ nA}$, the voltage on RA0 will fall at $200 \times 10^{-9} \div 100 \times 10^{-9} \text{ V/s} = 2 \text{ V/s}$.

The time needed for the voltage on RA0 to fall from 5 V to the low input threshold (VIL) of around 0.6 V would then be $(5 \text{ V} - 0.6 \text{ V}) / (2 \text{ V/s}) = 2.2 \text{ s}$.

Hence, with $C = 100 \text{ nF}$ and $I_{ULP} = 200 \text{ nA}$, the PIC should wake after around 2.2 seconds.

In practice, the prototype flashed every 5.3 s, so the sleep time was close to 4.3 s (the LED being on for 1 s in each cycle). This means that, in the 16F684 used in the prototype, I_{ULP} was very close to 100 nA.

This demonstrates that you should not use ULPWU where an accurately-timed wake-up is necessary (unless you perform measurements, possibly within your code, to calibrate the sleep period).

Of course, for the PIC to wake when the voltage on RA0 changes from 'high' to 'low', RA0 has to be configured as a digital input, and selected for interrupt-on-change (see [lesson 7](#)).

As mentioned earlier, to configure RA0 as a digital input, it has to be deselected as an ADC input, and we can deselect all ADC inputs by clearing the ANSEL register.

RA0 is also, by default, a comparator input (C1IN+). All of the comparator inputs can be disconnected, by selecting comparator mode 7, which is done by loading the value '7' into the CMCON0 register.

So, to make RA0 digital, we have:

```
banksel CMCON0          ; turn off comparators
movlw  0x07             ; (comparator mode 7)
movwf  CMCON0
banksel ANSEL           ; select digital mode
clrf   ANSEL           ; for all pins
                          ; -> RA0 in digital mode
```

To enable interrupt-on-change on RA0, we have:

```
; configure interrupt-on-change
banksel IOCA            ; enable interrupt-on-change
bsf    IOCA,0           ; on RA0
bsf    INTCON,RAIE     ; enable wake-up (interrupt) on PORTA change
```

Note that, as explained in [lesson 7](#), there is no need to set the global interrupt enable bit, GIE, because we are not actually using interrupts; we're only using an interrupt source to wake the device from sleep mode.

Finally, just before entering sleep mode, it is important to clear the port change interrupt flag, RAIF, to avoid an immediate false trigger:

```
bcf    INTCON,RAIF     ; clear port change interrupt flag

; enter sleep mode (until ULP wake-up)
sleep
```

Complete program

Here is how it all fits together:

```
;*****
;
; Description:      Lesson 10, example 3
;
; Demonstrates use of ultra low-power wake-up mode
; for periodic wake from sleep
;
; Turn on LED for 1 s, turn off, then sleep for approx 2 s
; (LED flashes for 1 s every 3 s (approx))
;
;*****
```

```

; Pin assignments:
; UPLWU (RA0) - 100 nf cap + 220R resistor
; RC0 - indicator LED
;
;*****

list      p=16F684
#include   <pl6F684.inc>

#include   <stdmacros-mid.inc>      ; DelayMS - delay in milliseconds

errorlevel -302      ; no "register not in bank 0" warnings
errorlevel -312      ; no "page or bank selection not needed" messages

radix     dec

EXTERN    delay10      ; W x 10ms 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    MCLRE_ON & _CP_OFF & _CPD_OFF & _BOD_OFF & _WDT_OFF & _PWRTE_ON
& _INTOSCIO & _FCMEN_OFF & _IESO_OFF

; pin assignments
constant  nLED=0      ; indicator LED on RC0

;*****
RESET     CODE    0x0000      ; processor reset vector

;***** Initialisation
; configure ports
banksel   PORTC      ; start with PORTC clear (LEDs off)
clrf     PORTC
banksel   TRISC      ; configure all PORTC pins as outputs
clrf     TRISC
banksel   CMCON0     ; turn off comparators
movlw    0x07        ; (comparator mode 7)
movwf    CMCON0
banksel   ANSEL      ; select digital mode
clrf     ANSEL       ; for all pins
; -> RA0 in digital mode

; configure interrupt-on-change
banksel   IOCA      ; enable interrupt-on-change
bsf      IOCA,0     ; on RA0
bsf      INTCON,RAIE ; enable wake-up (interrupt) on PORTA change

;***** Main loop
loop
; turn on LED
banksel   PORTC
bsf      PORTC,nLED

; charge ULPWU capacitor
banksel   TRISA      ; configure RA0 as an output
bcf      TRISA,0
banksel   PORTA      ; set RA0 high
bsf      PORTA,0

```

```
; delay 1 s
DelayMS 1000

; turn off LED
banksel PORTC
bcf     PORTC,nLED
; setup ULP wake-up
banksel PCON          ; enable ULPWU mode
bsf     PCON,ULPWUE
banksel TRISA        ; configure RA0 as an input
bsf     TRISA,0
bcf     INTCON,RAIF  ; clear port change interrupt flag

; enter sleep mode (until ULP wake-up)
sleep

; repeat forever
goto   loop

END
```

Although this lesson by no means exhausts all of the “enhanced” features of the PIC16F684, that’s enough for now. We’ll see a number of other features in later lessons.

In the meantime, we’ll pick up again where [lesson 9](#), on comparators, left off.

Now that we’ve introduced the PIC16F684, in the [next lesson](#) we’ll take a look at the 16F684’s dual comparator module.