Galen Sasaki, Department of Electrical Engineering, University of Hawaii

Explanation of PIC 16F84A processor data sheet -- Part 3: Final Overview of PIC

This is the last of the three part overview of the PIC processor. We will discuss interrupts, oscillators, reset and the sleep mode. Most of the topics are relevant to Lab 2.3. The topics that aren't as relevant are Reset and Oscillators, but these were included to round out the overview.

Interrupts

What is an *interrupt*? Here is an example to illustrate the concept. Suppose we have a computer that is normally used to play video games as shown in Figure 1. The computer also has Internet telephony software that makes it work like a telephone. The computer user would like to play the game until he/she gets a telephone call. Then the user would like to talk on the telephone. When the conversation is over, the user will resume playing the game. Now, the telephone connection acts like an *interrupt* (interruption) to the game, and when interrupted, the telephone software has to step in (*interrupt handler)*.

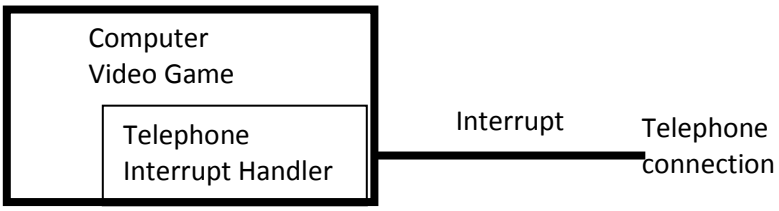


Figure 1. Computer that runs video games and a telephone.

Interrupts can be implemented by hardware or software. A software interrupt is similar to a function call in C. A hardware interrupt is also like a function call in C except that *invoking the call is done by hardware, i.e., a signal from a circuit, e.g., an external signal from a pin of the PIC*. Before explaining interrupts further, we will review a function call. The following is an example of a function and a function call.

```
int sum;  // Global variable

main()  {
int i;
sum = 1;
while(1) incr2();
}

void incr2()
{
sum++;
sum++;
 }
```

Note that *main* calls the function *incr2*. Also, note that *incr2* is called on every pass through the while-loop. Next, is an example of a program with an interrupt handler "clear".

```
int sum;  // Global variable

main()  {
int i;
sum = 1;
while(1) incr2();
}

void incr2()
{
sum++;
sum++;
 }

// ---- Interrupt handler
void interrupt clear()
{
sum = 0;
}
```

The *interrupt handler* clear( ) looks like a C function but it is never called by main( ) or any other C function in the program, e.g., incr2( ). This is an *interrupt handler* for a hardware interrupt. It is associated with a pin on the processor chip. Whenever the pin is enabled (causing an interrupt), the processor will jump from the main program and execute the interrupt handler. When it has finished executing the handler, the processor resumes executing the main program.

We will discuss the interrupt features of the PIC 16F648A, but let's first discuss a processor with a simpler interrupt system to illustrate some basic concepts. The processor is shown in Figure 2(a). In this example, whenever there is a positive transition on pin 2, it will cause an interrupt. The processor shown in Figure 2(b) has an enhancement. It has a software programmable bit called INTE for Interrupt Enable. Whenever INTE = 1, the interrupt works. When INTE = 0, then the interrupt is disabled. Basically, the INTE bit will cut off the INT signal through the AND circuit. This interrupt is referred to as "maskable" since because INTE can mask (in this case, cut-off) interrupt signals.

(a) **Figure 2.** Processors with simple interrupts. (b)

The interrupts in Figure 2 are called *edge triggered* because they occur whenever a signal transition occurs. There are also *level triggered* interrupts, where an interrupt occurs when a voltage value occurs, e.g., whenever the voltage value is 0v.

An issue with interrupts is that once they occur, you would like to complete service of the interrupt without interruption. Figure 3 shows one way to accomplish this. The internal interrupt circuitry of the processor is positive edge triggered.

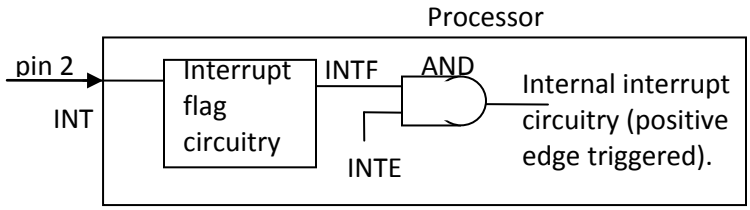


Figure 3. Another processor with a simple interrupt.

Also shown is an interrupt flag circuit between the external signal input and the AND circuit. This circuit's output is called INTF for interrupt flag. Normally, it's zero. But when an interrupt event occurs, the INTF will go to 1. This causes a positive edge that will trigger an interrupt (if INTE = 1). The INTF stays set at 1, any subsequent interrupt event does not cause a positive edge to the processor. The interrupt is effectively disabled.

The processor is designed so that the INTF flag can be cleared to zero by software. In this way the program can re-enable the interrupt pin once it has completed service of the current interrupt.

The interrupt flag circuitry will have different implementations depending on the type of interrupt, e.g., whether it is edge triggered or level triggered. But its main property is that once an interrupt event has occurred, the INTF is set to 1 and remains there until it is reset by software.

Figure 4 is an example of two interrupts at pins 1 and 2. The processor will interrupt if either of the two pins causes an interrupt. Both pins can be disabled through INT1E and INT2E. Also, both pins have interrupt flags INT1F and INT2F. Notice that the OR circuit indicates that an interrupt occurred but not which one. The processor must then check (or poll) the two flags INT1F and INT2F to determine which pin caused the interrupt. The flag that is set to 1 caused the interrupt. The program can now choose the right service to for the interrupt.

Figure 5 shows the interrupt logic for the PIC 16F648A. There are a lot of interrupt possibilities, but we will concentrate on a few, and in particular T0I (Timer0 circuit), INT (edge triggered interrupt of pin RB0/INT), and RBI (level triggered interrupts of PORTB).

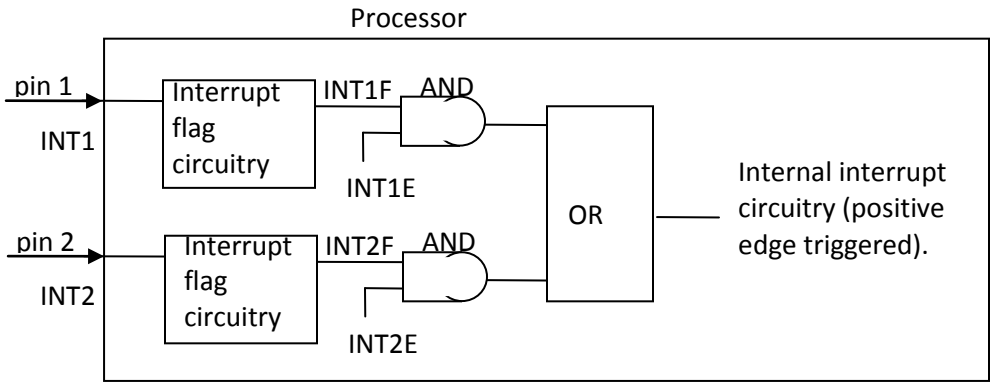
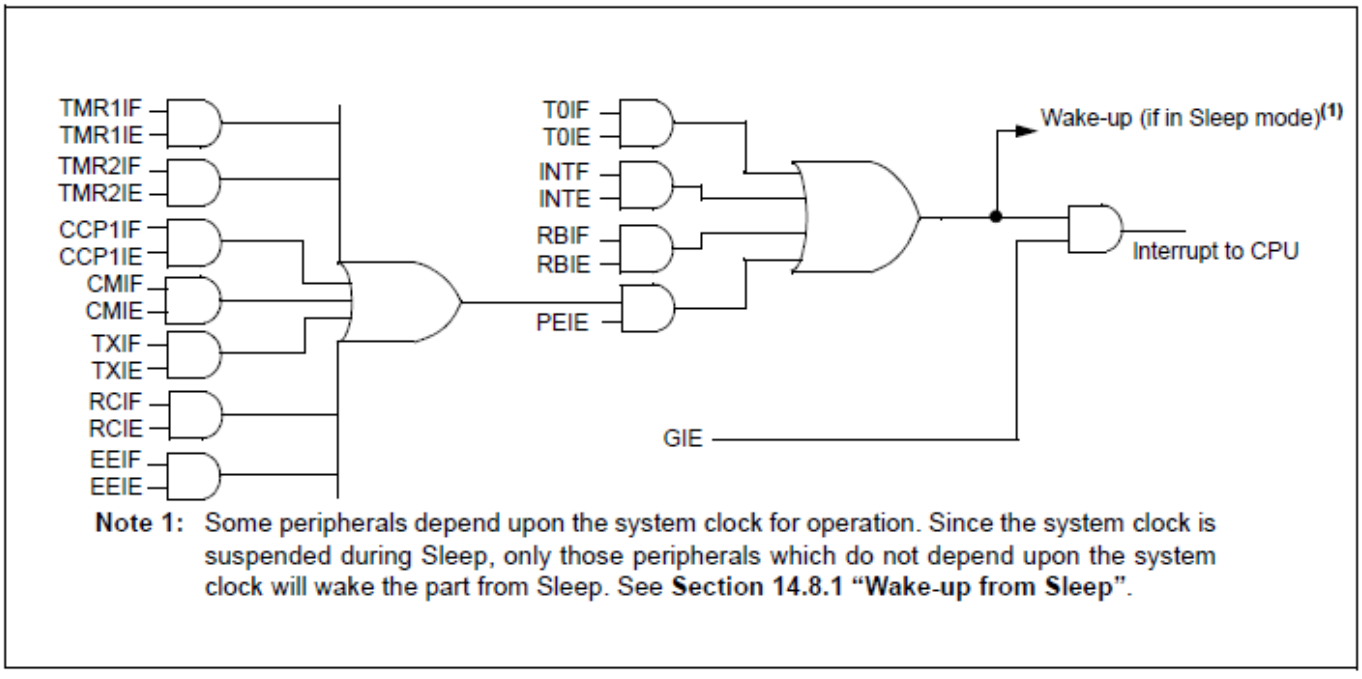


Figure 4. Yet another processor with a simple interrupt.

FIGURE 14-14: INTERRUPT LOGIC



Note 1: Some peripherals depend upon the system clock for operation. Since the system clock is suspended during Sleep, only those peripherals which do not depend upon the system clock will wake the part from Sleep. See **Section 14.8.1 "Wake-up from Sleep"**.

Figure 5. Interrupt logic of the PIC 16F648A from page 109 of the datasheet.

The PIC 16F648A has the INTCON register (interrupt control register). Figure 6 has a description. It also explains the bits in Figure 5.

REGISTER 4-3: INTCON – INTERRUPT CONTROL REGISTER (ADDRESS: 0Bh, 8Bh, 10Bh, 18Bh)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-x
GIE	PEIE	T0IE	INTE	RBIE	T0IF	INTF	RBIF
bit 7							bit 0

bit 7 **GIE:** Global Interrupt Enable bit
1 = Enables all un-masked interrupts
0 = Disables all interrupts

bit 6 **PEIE**: Peripheral Interrupt Enable bit
1 = Enables all un-masked peripheral interrupts
0 = Disables all peripheral interrupts

bit 5 **T0IE**: TMR0 Overflow Interrupt Enable bit
1 = Enables the TMR0 interrupt
0 = Disables the TMR0 interrupt

bit 4 **INTE**: RB0/INT External Interrupt Enable bit
1 = Enables the RB0/INT external interrupt
0 = Disables the RB0/INT external interrupt

bit 3 **RBIE**: RB Port Change Interrupt Enable bit
1 = Enables the RB port change interrupt
0 = Disables the RB port change interrupt

bit 2 **T0IF**: TMR0 Overflow Interrupt Flag bit
1 = TMR0 register has overflowed (must be cleared in software)
0 = TMR0 register did not overflow

bit 1 **INTF**: RB0/INT External Interrupt Flag bit
1 = The RB0/INT external interrupt occurred (must be cleared in software)
0 = The RB0/INT external interrupt did not occur

bit 0 **RBIF**: RB Port Change Interrupt Flag bit
1 = When at least one of the RB<7:4> pins changes state (must be cleared in software)
0 = None of the RB<7:4> pins have changed state

Figure 6. Interrupt Control (INTCON) register description from page 26 of the data sheet.

The GIE bit will enable or disable all the interrupts. Its default value is 0, which means all interrupts are disabled. Also, whenever the processor is interrupted, the GIE bit is cleared to 0. This ensures that while the processor can complete handling the interrupt without being interrupted again. When the program completes execution of the interrupt handler routine, the GIE bit is set to 1 again and the processor will subsequently accept new interrupt signals.

The other bits are interrupt flags and enable bits for the TIMER0, INT edge triggered interrupt, PORTB interrupts, and the peripheral interrupts. We will not use the peripheral interrupts in this lab, so the PEIE bit should be cleared to 0. Notice in Figure 5 that disabling PEIE will disable a lot of interrupts.

The TIMER0 interrupt occurs when the TIMER0 circuit wraps around, i.e., it goes from the value 255 to 0, which sets the T0IF flag to 1 – recall this from Lab 2.2.

Some of the bits of PORTB can also be interrupt inputs. An interrupt event occurs whenever the values change, from high-to-low or from low-to-high. We will not use these in Lab 2.3, so we should clear RBIE = 0.

We will use the INT edge triggered interrupt, so we should set INTE= 1. Notice that the RB0/INT pin is used as INT, and so RB0 is not available as a port for the PIC.

The following summarizes how we should set the bits of INTCON for Lab 2.3.

Bit #	Name	Value	Comment
7	GIE	1	Enable interrupts
6	PEIE	0	Disable peripheral interrupts
5	T0IE	0	Disable TIMER0 interrupt
4	INTE	1	Enable INT interrupt
3	RBIE	0	Disable RB interrupts
2	T0IF	x	Don't care since we disabled this interrupt
1	INTF	0	Clear INTF so we can accept a new interrupt from INT
0	RBIF	x	Don't care since we disabled this interrupt

Another register to consider for interrupts is the OPTION_REG, which we used earlier in Lab 2.2. This is shown in Figure 7. In particular the Interrupt Edge Select (INTEDG) bit can control whether the interrupt is positive or negative edge triggered. For Lab 2.3, we should set INTEDG = 1 for a positive edge triggered interrupt.

REGISTER 4-2: OPTION_REG – OPTION REGISTER (ADDRESS: 81h, 181h)

R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1	R/W-1
$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0
bit 7							bit 0

bit 7 **$\overline{\text{RBPU}}$**: PORTB Pull-up Enable bit
1 = PORTB pull-ups are disabled
0 = PORTB pull-ups are enabled by individual port latch values

bit 6 **INTEDG**: Interrupt Edge Select bit
1 = Interrupt on rising edge of RB0/INT pin
0 = Interrupt on falling edge of RB0/INT pin

Figure 7. The OPTION_REG from page 25 of the data sheet.

Here is a complete program for the PIC 16F648A that illustrates the use of an interrupt to reset a Counter.

```
#include <htc.h>

int count;  // Global variable that is the state of a counter
void backup(void);
//  The main routine constantly increments "count" and calls the function "back-up"

main()
{
//Check what is being initialized in the next statement.  Which interrupts are enabled and disabled?
INTCON=0b10010000;

// The following statement has the interrupt of INT occur on the rising edge.   Here we use a bit-wise OR operation.
// The value “01000000” is called a “mask”.  If a mask bit is 0 then the resulting bit stays the same, but if a mask bit
// is 1 then  the resulting bit is set to 1.  In this case, we want bit 6 (INTEDG) of OPTION_REG to be set  to 1, and all
// the other bits remain unchanged.
OPTION_REG = OPTION_REG | 0b01000000;

count = 0;

while(1) {
  count++; // Increments count 4 times
  count++;
  count++;
  count++;
  backup(); // Decrements count 3 times
  }
}

void backup()     // The function backup will decrement the "count" three times
{
count--;
count--;
count--;
}

void interrupt clear_count()  // This is the interrupt handler.  It clears "count".
{
count = 0;
INTF = 0; // Reenable the INT interrupt
}
```

If you want to read more about interrupts for the PIC 16F648A, the section in the data sheet is Section 14.5 that begins on page 109.

The above program can be made easier to read by using macros from header files for the PIC. If the pic16f648.h header is used then we can replace

```
OPTION_REG = OPTION_REG | 0b01000000;
```

with INTEDG = 1;

and

```
INTCON=0b10010010;
```

with

```
INTCON = 0;  // Clear all bits in INTCON
INTE = 1;      // Then set INTE and GIE
INTF = 0;      // To ensure this flag doesn't interfere with an interrupt signal
GIE = 1;       // After setting all the configuration bits of the interrupts, you can allow interrupt signals in
```

(Note that GIE is set to 1 at the end after all the interrupt enable bits have been set.)

To determine which of these labels can be used, read the pic16f648.h header file. Attached to this document is pic16f628.h, which is for a similar PIC though it has less program memory -- I couldn't find a file for the 16f648.

The header file can also be found at the following web site (this is easier to read):

http://www.koders.com/c/fidF0A44D2E07FDA885907A43D10570315D0D832304.aspx

Register names come under "Register Definitions", INTCON register comes under "INTCON bits", and OPTION_REG register comes under "OPTION_REG bits".

Also you can set the Configuration Bits in software too. Recall that so far we have set the Configuration Bits through MPLAB under the "Configure" window. But you can set the configuration bits using the command

```
_CONFIG(  );
```

which is a MACRO from the C compiler. List all the options you want for the configuration bits. For example, to turn off the Watchdog Timer, select the Crystal Oscillator (XT), and turn on the master clear, we have

```
_CONFIG(_WDT_OFF  &  _XT_OSD  &  _WDT_OFF);
```

But don't use this method for Lab 2.3, just use the configuration option in MPLAB as you did for the previous labs.

Reset

There are a number of options to reset the PIC. This is to ensure that it powers up properly. Also, there are different ways the PIC can be reset. Three possibilities are a time out of the watch dog timer, brown-out reset, and an external reset signal. Section 14.3 of the data sheet on page 101 explains these reset options.

Power Down Mode

An important feature of the PIC is the ability to go to sleep. In this mode, the PIC uses minimal current (power), and the internal clock is suspended, though the Watch Dog Timer is still running. While in sleep mode, the PIC suspends running the program; and when awaken, it resumes running the program. The PIC goes to sleep by executing the SLEEP instruction. It can be awaken in one of three ways: reset signal, Watch Dog Timer time-out, or interrupt from an external signal. It must be configured to wake up in one of these ways before it goes to sleep.

The reset signal will cause the PIC to reset, but the other two ways will cause the PIC to resume running from where it slept. Note from Figure 5, the interrupts will cause the PIC to awaken regardless of the value of GIE (general interrupt enable). However, the behavior of the PIC when it is awaken depends on the GIE bit. If the GIE bit = 0 then the PIC resumes running its program. If the GIE bit = 1 then the interrupt is enabled. Thus, when the PIC is awakened by an interrupt signal, it goes to the interrupt handler.

A detailed discussion of the power down mode is in Section 14.8 of the data sheet starting at page 112.

Watch Dog Timer

Recall that a Watch Dog Timer can help ensure that a processor avoids getting stuck in a bad state. When a time out occurs, the processor is reset. However, if the processor is in sleep mode then the processor just resumes executing the program at the time it went to sleep. So the processor behaves different to a time out depending on whether it is awake or asleep.

The Watch Dog Timer has a nominal time out of 18 ms. The timer can use the prescalar circuit as a *postscalar*. A postscalar circuit will lower the rate of the time out. The postscalar division ratios are 1:1, 1:2, ..., 1:128. For example, if we choose a 1:128 division ratio, the Watch Dog Timer will time out at 128 x 18 ms = 2.3 seconds. The postscalar can be enabled by setting the PSA bit to 1 in the OPTION_REG. The postscalar is also set in the OPTION_REG in the PS2-PS0 bits. Figure 8 shows the postscalar circuit with the Watch Dog Timer. Figure 9 has the values to set PS2-PS0.

CLRWDT is the machine instruction that clears the Watch Dog Timer. It also clears the postscalar, so that may have to be reinitialized.

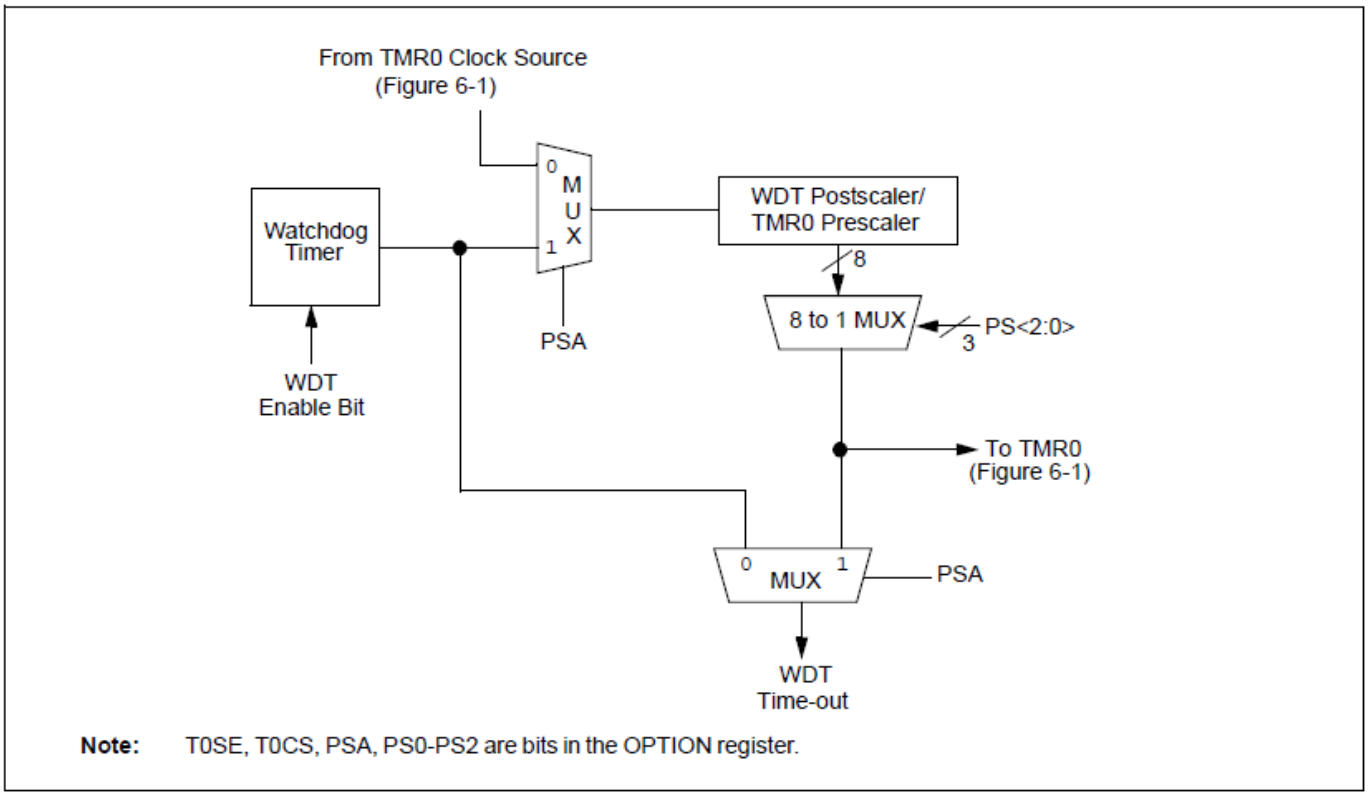
To access the CLRWDT and SLEEP instructions in a C program, you can use

```
#asm
CLRWDT
#endasm
```

or in Hitech C the macros are

```
SLEEP();
CLRWDT();
```

FIGURE 14-16: WATCHDOG TIMER BLOCK DIAGRAM



Note: T0SE, T0CS, PSA, PS0-PS2 are bits in the OPTION register.

TABLE 14-9: SUMMARY OF WATCHDOG TIMER REGISTERS

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Value on POR Reset	Value on all other Resets
2007h	CONFIG	LVP	BOREN	MCLRE	FOSC2	$\overline{\text{PWRTE}}$	WDTE	FOSC1	FOSC0	uuuu uuuu	uuuu uuuu
81h, 181h	OPTION	$\overline{\text{RBPU}}$	INTEDG	T0CS	T0SE	PSA	PS2	PS1	PS0	1111 1111	1111 1111

Legend: x = unknown, u = unchanged, - = unimplemented read as '0', q = value depends upon condition.
Note: Shaded cells are not used by the Watchdog Timer.

Figure 8. Watch Dog Timer and postscalar information from the data sheet on page 112.

PS<2:0>: Prescaler Rate Select bits

Bit Value	TMR0 Rate	WDT Rate
000	1 : 2	1 : 1
001	1 : 4	1 : 2
010	1 : 8	1 : 4
011	1 : 16	1 : 8
100	1 : 32	1 : 16
101	1 : 64	1 : 32
110	1 : 128	1 : 64
111	1 : 256	1 : 128

Figure 9. The PSA and PS2-PS0 values in OPTION_REG from page 25 of the data sheet.

When the Watch Dog Timer times-out then the \TO bit in the STATUS register is cleared to 0 (note that \TO should be TO overbar) as shown in Figure 10. When a PIC goes to sleep and then wakes up, it can check this bit to determine if it woke up due to the Watch Dog Timer.

:R 4-1: STATUS – STATUS REGISTER (ADDRESS: 03h, 83h, 103h, 183h)

R/W-0	R/W-0	R/W-0	R-1	R-1	R/W-x	R/W-x	R/W-x
IRP	RP1	RP0	$\overline{TO}$	$\overline{PD}$	Z	DC	C
bit 7							bit 0

bit 7 **IRP**: Register Bank Select bit (used for indirect addressing)
1 = Bank 2, 3 (100h-1FFh)
0 = Bank 0, 1 (00h-FFh)

bit 6-5 **RP<1:0>**: Register Bank Select bits (used for direct addressing)
00 = Bank 0 (00h-7Fh)
01 = Bank 1 (80h-FFh)
10 = Bank 2 (100h-17Fh)
11 = Bank 3 (180h-1FFh)

bit 4 $\overline{TO}$: Time Out bit
1 = After power-up, CLRWDT instruction or SLEEP instruction
0 = A WDT time out occurred

bit 3 $\overline{PD}$: Power-down bit
1 = After power-up or by the CLRWDT instruction
0 = By execution of the SLEEP instruction

Figure 10. STATUS register from page 24 of the data sheet of the PIC.

The following is a blinking LED program that sleeps and awakens using the Watch Dog Timer. Pin RA1 is connected to the LED.

```
main( ) // BLINKING LED program
{
// Set PORTA so that RA4-RA0 are outputs.
TRISA = 0b00000;

while(1) { // Loop forever
   RA1 = 0;          // Turn LED off
   delay10ms();      // Drive the output for 10 ms
   powerdown();      // Go to sleep for 1.2 seconds.  This function is on the
                       // next page.
   RA1 = 1;          // Turn LED on
   delay10ms();      // Drive the output for 10 ms
   powerdown();      // Go to sleep for 1.2 seconds
   }
}

void delay10ms( ) // A delay of (approximately) 10000 clock cycles
{
unsigned n;

TIMER0 = 0; // Initialize TIMER0.  This clears the prescalar.
// Now we set the OPTION_REG so that TOCS, TOSE, PSA, and PS2-PS1 are 0.
n = OPTION_REG & 0b11000000;  // The prefix “0b” means the number is binary
n = n | 0b00000111;
OPTION_REG = n;
while (TIMER0 < 40);
}
```

```
void powerdown() // Go to sleep for approximately 1.2 seconds
{
unsigned n;

n = OPTION_REG & 0b11110000;  // Mask out bits for PSA and PS2-PS0
n = n | 0b00001110;           // Set PSA = 1 (to switch prescalar to WDT)
                              // and PS2-PS0 to 110.  Then
                              // for the postscalar, the divide is 1:64.
                              // Then watch dog timer timeout = 1.8ms x 64
                              // which is 1.2 seconds
#asm
CLRWDT;                       // Clear watch dog timer, which also clears PS2-PS0
#endasm

OPTION_REG = n;               // Set PSA and PS2-PS0.

#asm
SLEEP;                        // Go to sleep
#endasm
}
```

Oscillators

The PIC processor requires a clock signal and it has a number of options. It can use a crystal as we have seen in the previous labs. It can use an internal clock circuit or it can use an external clock signal. The crystal configuration is the most accurate method of generating a clock signal.

If clock accuracy is not very important then the PIC's own internal clock generator circuit can be a good choice since it leads to a simpler circuit. This internal clock circuit is an RC circuit connected to a comparator (Schmidt trigger) as shown in Figure 11. When the input to the comparator has low voltage then the RC circuit will pull the input signal up. The rate that the signal rises depends on the RC value. When the input is above a threshold then the output of the comparator goes high. This causes the transistor to discharge the input of the comparator. The input goes back down to a low voltage and the comparator output goes low as well. The transistor is turned off, and again the RC circuit pulls the input of the comparator up again.

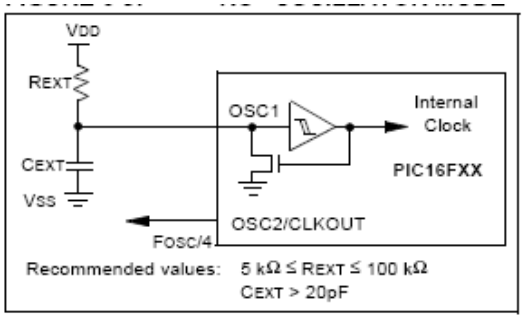


Figure 11. The RC oscillator configuration.

Odds and Ends

LVP is the low-voltage programming bit. It is part of the CONFIG register. If LVP = 1 then the low-voltage programming is enabled. We don't use this. So set LVP = 0. This will the RB4/PGM bit operate as the RB4 port. So if LVP = 1, we can't use the RB4 port.

Finally, you will notice that we also have PORTA ports (e.g., RA0, RA1, ...). Some of these ports are shared with the crystal oscillator (RA6 and RA7) and the master clear (RA5).

The rest of the ports are shared with analog comparator circuits (RA2 – RA5). The default of these ports is that they are disabled. To enable them you set bits in the Comparator Control (CMCON) and Voltage Reference Control (VRCON) registers. Basically, you have to set the last three bits of CMCON to 1s (ones) to disable the comparators and allow the pins to be connected to RA2-RA5. You must also set TRISA to configure the ports as inputs or outputs, where "1" means "input" and "0" means "output". This is similar to TRISB for PORTB.

We haven't used PORTA because it's more complicated than PORTB, and PORTB is sufficient for our needs.