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. (These features are discussed in more detail Section 6 in the PIC data sheet.)

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.](image)

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., an external signal from a pin. Before explaining interrupts further, we will review a function call. The following is an example of a function and a function call.

```c
int sum; // Global variable

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

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

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

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

1
int sum; //Global variable

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

void incr2() {
  sum++;
  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.

On the PIC 16F84A, one hardware interrupt is from the pin RBO/INT. This pin can either be the IO port RBO or the interrupt input INT. Configuring this pin is done by setting values in the INTCON and OPTION_REG registers. Here are the configuration options:

- Pin RBO/INT is either IO port (RBO) or interrupt (INT)
- The interrupt is enabled on the rising or falling edge of the signal
- The interrupt input is enabled or disabled.

Figures 2 and 3 show the bits of INTCON (Interrupt control) and OPTION_REG, respectively. For INTCON, notice that some bits are Enable bits, and other bits are Flags. You can set values of Enable bits to control the interrupts. For example, GIE is the Global Interrupt Enable. Setting it to 1 will enable all (unmaskable) interrupts. Clearing it to 0 will disable the interrupts. In our case, we want GIE = 1.

Another enable is INTE, the external interrupt enable. We want to set INTE = 1. Now INTF is the interrupt flag for the external hardware interrupt. This indicates when such an interrupt occurs. Thus, your program can check when the interrupt occurred. You can clear this bit to 0 by writing to it.

Figure 2 shows enabling and flag bits for other interrupts. These include an internal hardware interrupt caused by the TIMER0 timer module. When the module changes value from ffh to 00h, it can cause an interrupt. Also, whenever
PORTB changes value, an interrupt can occur. Each of these interrupts can cause the processor to go to the interrupt handler. If you want these different types of interrupts to be handled in different ways then the handler should check the interrupt flags. This will tell it which type of interrupt occurred.

<table>
<thead>
<tr>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-0</th>
<th>R/W-x</th>
</tr>
</thead>
<tbody>
<tr>
<td>GIE</td>
<td>EEIE</td>
<td>T0IE</td>
<td>INTE</td>
<td>RBIE</td>
<td>T0IF</td>
<td>INTF</td>
<td>RBIF</td>
</tr>
</tbody>
</table>

bit 7 GIE: Global Interrupt Enable bit
1 = Enables all unmasked interrupts
0 = Disables all interrupts

bit 6 EEIE: EE Write Complete Interrupt Enable bit
1 = Enables the EE Write Complete interrupts
0 = Disables the EE Write Complete interrupt

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 = At least one of the RB7:RB4 pins changed state (must be cleared in software)
0 = None of the RB7:RB4 pins have changed state

Figure 2. INTCON bit values.

Figure 3 shows the bits of the OPTION_REG register. There is one bit associated with interrupts which is INTEDG (Interrupt Edge Select). This bit controls whether the INT input triggers an interrupt by a rising or falling signal edge. It can be set to 0 or 1 through software.

Here are more details about interrupts. Consider the program above where “clear” is the interrupt handler. When the associated hardware interrupt occurs, the processor jumps to execute the interrupt handler. At this time, the GIE bit is cleared to 0 (recall the GIE bit is shown in Figure 2). This prevents any additional interrupts while the handler “clear” is being executed. In this way, the handler can finish executing without interruption. When the processor has finished executing the handler, the processor returns to the main program and the GIE is set to 1, allowing new interrupts.
Figure 3. The bits of OPTION_Reg.
Here is a complete program for the PIC 16F84A that illustrates the use of an interrupt to reset a Counter.

```c
#include <pic1684.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 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
}

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

// This is the interrupt handler. It clears "count".
void interrupt clear_count()
{
    // Since all interrupts are disabled except for the external interrupt INT, we don’t have to check interrupt flags.
    // We don’t have to clear flags either.
    count = 0;
}
```
Oscillators

The PIC processor requires a clock signal and it has a number of options. It can use a crystal (as shown in Figure 4) or it can use an external clock signal. The crystal configuration is the most accurate method of generating a clock signal. Capacitors should be selected as shown in Figure 4 to give a stable clock generation.

If clock accuracy is not very important then the PIC has its own internal clock generator circuit that does not require a crystal shown in Figure 5. This circuit is an RC circuit connected to a comparator (Schmidt trigger). 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 4.** The crystal oscillator configuration and capacitance values for stable performance.

**Figure 5.** The RC oscillator configuration.
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. Two possibilities are a time out of the watch dog timer or an external reset signal. Section 6 of the data sheet explains these reset options.

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 awakened 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 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 $128 \times 18 \text{ ms} = 2.3 \text{ seconds}$. 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 6 shows the postscalar circuit with the Watch Dog Timer. Figure 7 has the values to set PS2-PS0.

![Figure 6. Watch Dog Timer and postscalar.](image)

Note: PSA and PS2-PS0 are bits in the OPTION_REG register.
Figure 7. The PSA and PS2-PS0 values in OPTION_REG.

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

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

  while(1) { // Loop forever
    PORTA = 0b00000; // Output 0 to RA1
    delay10ms();     // Drive the output for 10 ms
    powerdown();     // Go to sleep for 1.2 seconds
    PORTA = 0b00001; // Output 1 to RA1
    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 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
  CLRWD;       // Clear watch dog timer, which also clears PS2-PS0
  #endasm

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

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

This is the end of the three part document describing the PIC microprocessor. We did not have time to cover all aspects, e.g., programming the EEPROM. But we did cover most of the features. When you have the time, you are encouraged to look over the entire document.