Using Interrupts

Introduction. This application note shows how to use the interrupt capabilities built into the PIC 16Cxx series controllers with a simple example in TechTools assembly language.

Background. Many controller applications work like a fire department. When there's no fire, they mend hoses, polish the fire truck, and wait for something to happen. When the fire bell rings, they swing into action and handle the emergency. When it's over, they return to waiting.

If there were no fire bell, the job would be much different. The fire crew would have to go out and look for fires, and somehow still make sure that the hoses were fixed and the truck maintained. In their scurry to get everything done, they might respond late to some fires, and miss others completely.

The newer PIC 16Cxx controllers have interrupts, which work like the fire bell in the first example. When an interrupt occurs, the PIC stops what it's doing and handles the cause of the interrupt. When it's done, the PIC returns to the point in the program at which it was interrupted.

The second example, with no fire bell, is an example of polling. This is the approach used with the 16C5x PICs, which lack interrupts. For a fast PIC, polling isn't nearly as bad as in the example. The PIC can often get everything done with plenty of time to spare. But there are times when interrupts are the simplest way to do two or more things at once.

How interrupts work. For the examples here, we're going to talk about the 16C84. The same principles apply to the other interrupt-capable PICs, but registers and interrupt sources may vary.

First of all, the 16Cxx PICs awaken from power-up with all interrupts disabled. If you don't want to use interrupts, don't enable them. It's that simple.

The PIC 16C84 can respond to interrupts from four sources:

- A rising or falling edge (your choice) on pin RB0/INT.
- Changing inputs to pins RB4 through RB7.

- Timer (RTCC) overflow from 0FFh to 0.
- Completion of the data EEPROM programming cycle.

You can enable any combination of these sources. If you enable more than one, it will be up to your code to determine which interrupt occurred and respond appropriately. More on that later.

Let's take a simple example. We want to use the RTCC interrupt to generate a steady 1-kHz square wave on pin ra.1. Every 500 µs the RTCC will interrupt whatever the PIC is currently doing, toggle ra.1, reload the RTCC with an appropriate value and return.

The first consideration is where to put the interrupt-handler code. When the 16C84 responds to an interrupt, it jumps to location 04h in program memory. So we'll use an **org** statement to place the handler at 04h. If you look at the program listing, you'll see that we actually have two **org**s: the first positions the code **jmp start** at 0, which is where the '84 looks for its power-on startup code, and the second positions the interrupt handler at 4.

The program's startup code configures the RTCC for its initial 500-µs timing period, and enables the interrupt. To do so, it turns on the first two bit switches shown in figure 1. For any interrupt to occur, the global-interrupt enable (GIE) bit must be set. For the timer interrupt to occur, the RTCC interrupt enable (RTIE) bit must be set. Once those two switches are closed, only one switch remains before the interrupt "alarm bell" goes off—the RTCC interrupt flag (RTIF).

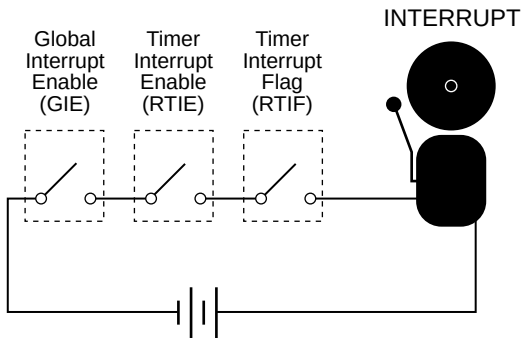


Figure 1. Logic of the interrupt-enable and flag bits for the RTCC.

When the interrupt occurs, the PIC clears GIE to disable further interrupts, pushes the program counter onto

the stack, then jumps to location 4. This process takes four instruction cycles for an internal interrupt like the RTCC, five for an external interrupt like RB0/INT.

Once at location 4, the PIC executes the code there until it encounters a **ret**, **retw**, or **reti** instruction. Any of these will pop the value from the top of the stack and cause a jump back to the point at which the program was interrupted. However, the normal way to return from an interrupt is the **reti** instruction. It automatically re-enables interrupts by setting GIE. The other two returns do not.

In the program listing, you will notice that the first thing the interrupt handler does is clear RTIF. Whenever the RTCC rolls over from 0FFh to 0, the PIC automatically sets RTIF. It does this regardless of the state of the interrupt bits. And it never turns RTIF back off. So it's the responsibility of the interrupt handler to clear RTIF. This is true of all of the interrupt flags. If the handler doesn't clear the appropriate flag, the same interrupt will occur again as soon as interrupts are re-enabled. The resulting endless loop is what the Microchip documentation calls "recursive interrupts."

With a single interrupt source, you can think of an interrupt as a hardware version of the **call** instruction. The primary difference is that this kind of **call** can occur anywhere in your program, whether or not the program is ready for it. This can pose a problem. Let's say your program is executing the TechTools instruction **add sum,delta** when the interrupt occurs. That **add** instruction actually consists of two instructions; one that loads the variable **delta** into the **w** register, and a second that adds **w** to **sum**.

Imagine that right after **delta** is loaded into **w**, the interrupt takes over. It performs, for instance, the instruction **mov rb,data**. This instruction loads the variable data into **w**, then moves **w** into **rb**. When it's done, **w** contains a copy of **data**.

When the handler returns, the PIC completes the interrupted addition. But **w** contains **data**, not the intended **delta**, causing an error.

There are two ways to prevent this. One is to disable interrupts before

a two-part instruction, then enable them again afterwards. If the interrupt event occurs while interrupts are disabled, the PIC will set the interrupt flag, but won't jump to the handler until your program re-enables interrupts. The PIC will not miss the interrupt, it will just be slightly delayed in handling it. To use this method, the example above would be changed to:

```
clrb    GIE         ; Interrupts disabled.
add     sum,delta   ; w = delta: sum = sum+w.
setb    GIE         ; Interrupts enabled.
```

The alternative approach is to begin your interrupt handler with code that saves a copy of the **w** register. Then, just before **reti**, move the copy back into **w**.

This takes care of compound instructions that use **w**, but leaves another group of instructions vulnerable; the ones that use the status register. Conditional jumps and skips like jump-if-zero (**jz**) and compare-and-jump-if-equal (**cje**) and their many cousins are probably obvious. More subtle are the rotate instructions **rr** and **rl** (which pull the carry bit into the most- or least-significant bit of the byte being rotated). If the interrupt handler affects any of the status bits, it may alter the outcome of the interrupted instruction.

To protect **w** and **status**, you must make copies of them at the beginning of a handler, then restore those copies at the end. In order to prevent the action of restoring **w** from affecting **status**, you can use a sneaky trick. Moving a file register into **w** will set or clear the zero bit, but moving a nibble-swapped copy of the register into **w** does not. Neither does swapping the nibbles of a file register. The program listing shows how to use these loopholes to accurately copy both **w** and **status**.

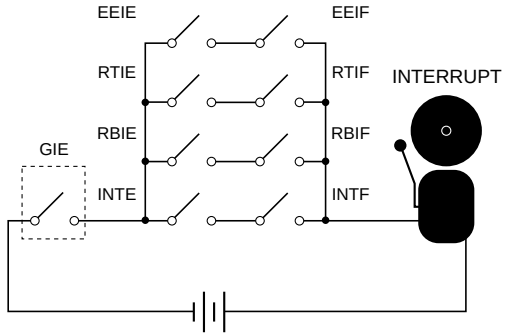
Keep your interrupt handlers as short and simple as possible. Examine them carefully for their effect on other portions of the program that might be executing at the time the interrupt occurs. If necessary, protect sensitive code by bracketing it with instructions that temporarily disable interrupts. Also, keep in mind that test running your program may not catch all possible interrupt-induced bugs. Use the PSIM simulator and a sharp eye to detect potential problems.

Once you understand how the mechanism works with a single interrupt source, you'll be relieved to know that it's not that much more difficult to handle multiple interrupt sources. Figure 2 shows the alarm-bell diagram for multiple interrupts on the 16C84. If you have all of the 16C84's interrupts enabled, your handler at 04h should begin with something like:

```
jb   INTF,  rb0_edge     ; If INTF then handle rb.0 interrupt.
jb   RBIF,  rb_change    ; If RBIF then handle change on rb.4-7.
jb   RTIF,  timeout      ; If RTIF, then handle timer rollover
jb   EEIF,  EE_wr_done ; If EEIF, then handle write complete.
```

Code at each of those labeled locations (**rb0_edge**, etc.) would then deal with that particular type of interrupt. Remember that each of the handlers must clear its corresponding flag; for example, **rb0_edge** must include the instruction **clrb INTF**.

What happens if an interrupt occurs while the PIC is already handling an interrupt? At that time, nothing. Remember that the PIC clears GIE automatically in response to an interrupt, then sets it when **rti** executes. Any interrupt event that occurs in the meantime will set the appropriate flag. That interrupt will be delayed until after the current one is finished.



INTE/F = RB0 interrupt enable/flag bits
RBIE/F = RB4Ð7 change interrupt enable/flag bits
RTIE/F = RTCC overflow interrupt enable/flag bits
EEIE/F = EEPROM write complete interrupt enable/flag bits

Figure 2. Logic of all four 16C84 interrupts.

Program design. Interrupts are not a cure-all for the difficulties of handling multiple tasks. In fact, you may just end up trading a difficult

programming job for an even more difficult debugging job. Experts suggest using interrupts only as a last resort.


```
;Program: INTRPT.SRC (Interrupt demonstration)
; This program illustrates the use of the RTCC interrupt
; in the PIC 16C84. The foreground program blinks an LED on
; pin ra.0, while an interrupt task outputs a 1-kHz square
; wave through pin ra.1 (assuming a 4-MHz clock).

; Device (16c84) and setup options.
          device  pic16c84,xt_osc,wdt_off,pwrt_off,protect_off

; Equates for LED, tone pins. Connect the LED through a
; 220-ohm resistor. Connect a speaker or earphone through
; a 1-k resistor.
LED       =    ra.0
SPKR      =    ra.1
tone      =    6  ; Load into RTCC for 500-us delay.

; Allocate space for some variables. Notice that in the '84
; variable start at 0Ch—higher than in the 16C5x series.
          org  0Ch
w_copy    ds   1
s_copy    ds   1
countL    ds   1
countH    ds   1

; On startup, the PIC looks at address 0 for its first
; instruction. Since the interrupt handler begins at
; address 4, we'll just jump over it to get to the
; startup routine.
          org     0
          jmp     start          ; Beginning of main program.

; Next is the interrupt handler, which must begin at
; address 4. This handler copies restores w and the status
; register. Because a normal "mov w,fr" alters the z bit of
; the status register, this routine uses "mov w,<>fr," which
; does not. The routine actually swaps the byte twice,
; resulting in the correct value being written to w without
; affecting the z bit.
          org     4
handler
          clrb    RTIF           ; Clear the timer interrupt flag.
          mov     w_copy,w       ; Make a copy of w.
          mov     s_copy,status  ; Make a copy of status.
```

```
        XOR   ra,#2             ; Toggle bit ra.1.
        mov   rtcc,#tone        ; Reload rtcc for 500-us delay.
        mov   status,s_copy     ; Restore status register
        swap  w_copy            ; Prepare for swapped move.
        mov   w,<>w_copy        ; Swap/move to w, status unaffected.
        reti                    ; Return to main program.

; Here's the startup routine and the main program loop.
; In the line that initializes "intcon," bit 7 is GIE and bit 5
; is RTIE. Writing 1s to these enables interrupts generally (GIE)
; and the RTCC interrupt specifically (RTIE).

Start
        mov   !ra,#0            ; Make ra pins outputs.
        setb  rp0               ; Switch to register page 1.
        clr   wdt               ; Assign prescaler to rtcc.
        mov   option,#0         ; Set prescaler to divide by 2.
        clrb  rp0               ; Restore to register page 1.
        mov   intcon, #10100000b    ; Set up RTCC interrupt.

; If the interrupt handler were to alter w, the LED would stop
; flashing or flash erratically, since the routine is written to
; rely on the value of w remaining 1 in order to toggle bit ra.0.
; The routine also relies on reliable operation of the status
; register because of the two skip-if-not-zero (snz) instructions.
; Although this structure is a little strange, it's an effective
; canary-in-a-coalmine demonstration that the interrupt handler's
; save/restore instructions do preserve both w and status.

        mov   w,#1              ; Bit in 0 position to toggle ra.0.
:loop   inc   countL            ; countL=countL + 1
        snz                     ; IF countL=0,
        inc   countH            ; THEN countH=countH+1
        snz                     ; IF countH=0,
        XOR   ra,w              ; THEN toggle LED.
        jmp   :loop
```