Lesson 20
Interrupts

Overview

Introduction

In many of our earlier examples, the PIC has spent most of its time polling some input or timer. Interrupts allow us to do useful work, and be notified when some event occurs that we care about.

In this section

Following is a list of topics in this section:

Description	See Page
Overview	2
The Timer Interrupt	5
Saving the count to EEPROM	8
Multiple Interrupt Sources	11
Additional Experiments	16
Wrap Up	17

Overview

Introduction

In many applications we want the PIC to "walk and chew gum" at the same time. By periodically polling inputs or polling timers, the PIC can appear to carry out multiple tasks. However to be responsive to rapidly occurring events, the polling must be done very frequently. But more frequent polling means fewer compute cycles available for the "background" task. Using interrupts allows fast response to external events without wasting compute time on polling.

In theory, an interrupt is fairly simple. When an event happens that needs immediate attention, the interrupt taps the processor on the shoulder to go off and handle this new task. Once that task is done, the processor can return to what it was doing.

Mechanics

The mechanics of an interrupt are simple in concept. When some hardware event causes an interrupt; the current program counter is pushed onto the stack. Then the program counter is loaded with the address of the interrupt service routine. In the PIC16 family, this address is H'04'.

In practice, more than just the program counter must be saved and restored. If the interrupt service routine does much of anything, it will probably change the STATUS and "W" registers. Thus we need to save the values in those registers before proceeding with interrupt processing. The interrupt may also call on other assets within the PIC which also must be properly saved and restored. This is the most important, and sometimes the most difficult, part of interrupt processing

Interrupt Sources

The PIC16F84A has only 4 interrupt sources:

Name	Function	Enable Bit	Flag Bit
RB0/INT	A transition of the RB0 pin	INTE	INTF
TMR0	An overflow of the TMR0 register	T0IE	T0IF
PORTB	A change in PORTB bits 4-7	RBIE	RBIF
EEPROM	Completion of an EEPROM write	EEIE	EEIF

Other PICs have additional interrupts, appropriate to the peripherals contained on those PICs. Note that bit names ending in IE are **I**nterrupt **E**nable, those ending in IF are **I**nterrupt **F**lags.

Controlling the interrupts

The INTCON register contains a bit, GIE (**G**lobal **I**nterrupt **E**nable), which enables all interrupts. As long as GIE is false, no interrupt from any source will occur. INTCON also contains a bit to enable each of the individual interrupts as shown in the table above. To receive an interrupt, both the GIE bit and one of the interrupt enable bits must be set.

When the interrupt does occur, a flag is set to indicate the specific source. These flags are also shown in the table.

Continued on next page

Overview, Continued

Saving Context

When an interrupt occurs, a number housekeeping chores must be done. The PIC hardware automatically saves the program counter. The hardware also clears the global interrupt enable to prevent a second interrupt from occurring while the first is being processed. When a return from interrupt instruction, (`retfie`) is executed, the program counter is automatically restored and the GIE is turned back on.

However the programmer has certain responsibilities. The specific flag that caused the interrupt must be cleared to prevent the same interrupt from being reprocessed. The programmer must also determine which registers must be saved and restored so that the main program can pick up where it left off. Remember that an interrupt can occur at any place in the main program, so all possibilities must be covered.

The STATUS and W registers are most often saved and restored in an ISR. Since doing almost anything changes the STATUS, this can be tricky. The solution is a bit of boilerplate code that begins and ends most interrupt routines.

```
Isr:
    movwf   w_temp              ; Save off the W register
    swapf   STATUS,W            ; And the STATUS (use swapf
    movwf   status_temp         ; so as not to change STATUS)
```

[Interrupt handling code goes here]

```
    bcf    (interrupt flag)
    swapf   status_temp,W       ; Restore the status
    movwf   STATUS              ; register
    swapf   w_temp,F            ; Restore W without disturbing
    swapf   w_temp,W            ; the STATUS register
    retfie
```

Since the `movwf` instruction doesn't influence the status, the "W" can be saved directly. Next the `swapf` allows moving the contents of the STATUS register into W without affecting the status bits. The nybbles of STATUS are reversed in the saved version, but get reversed again in the restore process.

The STATUS register is restored with a simple `movwf`. However restoring W this way would affect the status bits. So the `swapf` trick is used again, this time twice. The first time to get the nybbles backward so the second `swapf` ends up with the correct result in W.

There may be other registers that must also be saved and restored depending on what assets of the PIC the ISR uses. These might include the FSR and PCLATH. Also note that we must clear the flag for the interrupt we just handled. If we don't the interrupt will re-assert itself as soon as we return from the ISR.

Continued on next page

Overview, Continued

Interrupt Priority

On complex instruction set processors, there are usually a number of interrupt priorities; that is, an interrupt can itself be interrupted by a higher priority interrupt. On all the PIC16Fxx and smaller parts, all interrupts have the same hardware priority, but the effective priority can be managed by the software.

When an interrupt occurs, the software must determine which device caused the interrupt by examining the interrupt flags. The order in which this is done essentially determines the priority of an interrupt.

In many applications, especially on the PIC16F84, there will only be one interrupt enabled; in this case, the checking of other interrupt flags is normally skipped.

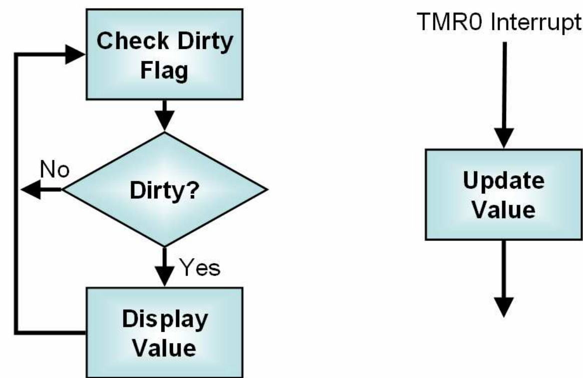
The Timer Interrupt

Introduction

For our first experiment, we will examine the TMR0 interrupt. We have seen in earlier lessons how we can use the TMR0 register to keep track of elapsed time while we are off doing something else. The TMR0 interrupt, when enabled, causes an interrupt whenever the TMR0 register overflows.

Overall view of our Program

Our program will consist of a tight loop that checks a "dirty" flag to see whether a value in memory has been updated. If the flag is set, we will display the value, otherwise we will go back and check the flag again:



We will rely on the LCD routines from Lesson 17 as well as the decimal conversion routine from Lesson 18.

Displaying a 16-bit Value

Because all of the experiments in this lesson will require displaying a 16 bit value on the LCD, it makes sense to make a subroutine to do that which we can simply call in each of our programs.

The ConvBCD2 routine takes a two byte value and returns a 5 byte result. To display this, we will need to loop through each of the 5 characters, sending them to the LCD. Since the same number of characters are sent each time, the LCD need not be erased. Instead we can simply set the LCD cursor position, which is slightly faster. Faster is good because we would like to be able to update the values quickly, and the flashing caused by erasing the display is a little annoying.

Continued on next page

The Timer Interrupt, Continued

Displaying a 16-bit Value (continued)

The code for the display routine will then look something like this:

```
Disp16:
            call   ConvBCD2      ; Convert the value to BCD
            call   LCDzero       ; Cursor to left of LCD
            movlw  digits        ; Pick up address of output
            movwf  FSR           ; into FSR
            movlw  H'05'         ; Count of number of
            movwf  count         ; digits to display
Disp16L
            movf   INDF,W        ; Get current digit
            call   LCDletr       ; Display it
            incf   FSR,F         ; Point to next digit
            decfsz count,F       ; Count down one we just did
            goto   Disp16L       ; Done? No, do it again
            movf   LEDflg,W      ; Set the LEDs to the
            movwf  PORTB         ; desired condition
            clrf   dirty         ; Value is now current

            return
```

Because the LCD and the LEDs share pins on the PIC-EL, the LEDs flash when the LCD is updated. To reduce that flashing, we will turn off the LEDs every time we write the LCD. This write to PORTB doesn't affect the LCD since we are careful not to raise the LCD enable pin. Instead of simply storing a literal, variable `LEDflg` is used so that later we can use the LEDs as an output.

Also note that we have taken care to clear the "dirty" flag after the write. Our ISR will set the dirty flag whenever the value is changed. In this way, the LCD is rewritten whenever the data changes, and only when the data changes.

The main program loop

The main program loop is pretty straightforward. The value is displayed on the LCD, then spin on the dirty flag until it becomes true, at which time we will display the value again:

```
Loop
            call   Disp16        ; Display the value in memory
Loop1
            movf   dirty,W       ; Test whether a new value
            btfsc  STATUS,Z
            goto   Loop1         ; No, check again
            goto   Loop          ; Yes, go do display
```

Initialization

The LCD is initialized, along with the values for the dirty flag, the binary value to count, and the `LEDflg`. Since the timer initialization will be reused, it is put in a separate routine. That initialization is essentially the same as in Lesson 13.

After initialization, it is now safe to turn on interrupts. Both the timer interrupt and the global interrupt enable must be turned on:

```
        bsf    INTCON,T0IE   ; Allow timer interrupt
        bsf    INTCON,GIE    ; Enable interrupts
```

Continued on next page

The Timer Interrupt, Continued

The Interrupt Service Routine

The interrupt service routine is responsible for incrementing the counter. The routine gets called whenever the timer overflows and causes an interrupt. The routine must increment the counter and set the dirty flag.

However, there are also some "housekeeping" tasks. Since we have no idea of the processor's state at the time the interrupt occurred, the STATUS and W registers must be saved before the ISR changes them. Of course, these must be restored on exit.

Since the state of the bank bits when the interrupt occurred is unknown, we need to select the bank before accessing any file registers, even if they are in bank 0. On the F84 this is only necessary for special function registers; the general purpose registers are accessible in all the banks. So far, all the registers we use in the ISR are also in all banks. However, later this could become an issue, and is one of the potential traps of interrupt handling. Finally, the timer interrupt flag must be cleared before exiting; otherwise the timer interrupt will re-assert itself the instant we try to exit the ISR.

Saving the status is done in a somewhat "rote" fashion:

```
IRQSVC code
        movwf  w_temp        ; Save off the W register
        swapf  STATUS,W      ; And the STATUS
        movwf  status_temp   ;
```

The actual work we need to do in the ISR is pretty basic:

```
        ; Bump up the two-byte value we will display
        incf   binary+1,F    ; Increment low byte
        btfsc  STATUS,Z      ; Overflow? (incf doesn't affect C)
        incf   binary,F      ; Increment high byte
        incf   dirty,F       ; Note that value changed
```

The restore is again pretty much copied from the datasheet, except we must remember to clear the interrupt flag

```
        bcf    INTCON,T0IF ; Clear the old interrupt

        swapf status_temp,W ; Restore the status
        movwf STATUS        ; register
        swapf w_temp,F      ; Restore W without disturbing
        swapf w_temp,W      ; the STATUS register
        retfie
```

Testing the code

Once we build and assemble the project, and run the code in our PIC-EL, we should see a five digit number incrementing quite quickly. Wait until the count exceeds 255 to be sure we have handled incrementing the second byte of `binary` correctly.

Saving the count to EEPROM

Introduction

In an earlier lesson, we examined reading the EEPROM. However, we avoided the problem of writing the EEPROM. It turns out that the EEPROM write is one of those interrupt sources we want to examine, so now we will look at writing the EEPROM.

The EEPROM is often used to store calibration constants, some of which may be quite important or difficult to recover. Because of this, it is important that a program not accidentally alter the contents of EEPROM. To make it difficult to inadvertently write to EEPROM, Microchip has the programmer "jump through hoops" to actually perform the write.

Writing to EEPROM isn't difficult, but there are several steps whose only purpose is to prevent an accidental write.

The EEPROM writing sequence

To write to the EEPROM, the following sequence of steps must be executed:

- Check that a write is not in progress (EECON1 bit WR is clear)
- Write the desired EEPROM address to EEADR
- Write the data value to be stored to EEDATA
- Set the WREN bit of EECON1 to enable writing
- Disable interrupts if not already disabled
- Write a H'55' to EECON2
- Write a H'AA' to EECON2
- Set the WR bit of EECON1
- Enable interrupts (if being used)
- Clear the WREN bit of EECON1

Note that the steps between turning off interrupts and turning them back on must happen in the exact sequence with no interruption.

The write can take some time, a few milliseconds depending on the supply voltage and data to be written. When the write is complete, the WR bit will be cleared, and EEIF will be set. If interrupts are enabled and EEIE is set, an interrupt will occur.

Continued on next page

Saving the count to EEPROM, Continued

The EEPROM writing code

The above sequence, in the case where we are not using the EEPROM write completion interrupt, would look like the following:

```
        banksel     EECON1      ; Bank 1
Wait1   btfsc       EECON1,WR   ; Be sure no write in progress
        goto        Wait1       ;
        banksel     EEADR       ; Bank 0
        movlw       SAVADR      ; Set address in EEPROM
        movwf       EEADR       ; to be written
        banksel     binary      ;
        movf        binary,W    ; Set data to be written
        banksel     EEDATA      ;
        movwf       EEDATA      ;

        banksel     EECON1      ; Bank 1
        bcf         INTCON,GIE  ; Turn off interrupts
        bsf         EECON1,WREN ; Enable write
        movlw       H'55'       ; This sequence is
        movwf       EECON2      ; required before the
        movlw       H'AA'       ; EEPROM may be written
        movwf       EECON2      ;
        bsf         EECON1,WR   ; Start write
        bsf         INTCON,GIE  ; Turn interrupts back on
Wait2   btfsc       EECON1,WR   ; Wait for write complete
        goto        Wait2
        bcf         EECON1,WREN ; Disable write
```

We have predefined a constant, SAVADR, that identifies the address in EEPROM where we want to store the value. In the example code, this is in an include file.

In our example, the above code needs to be duplicated for the second byte, since we have two bytes of count to store.

In this example, we really didn't require the **`banksel binary`**, although on other PICs, this is a real trap, so it isn't a bad practice to ensure bank 0 before accessing our variable. Having selected bank 0, the **`banksel EEDATA`** is also redundant, although it would have been required had we eliminated the **`banksel binary`**.

Reading the EEPROM

Reading the EEPROM is fairly simple; we did it back in Lesson 14. We simply store the address to read in EEADR, set the RD bit in EECON1, and read the result from EEDATA.

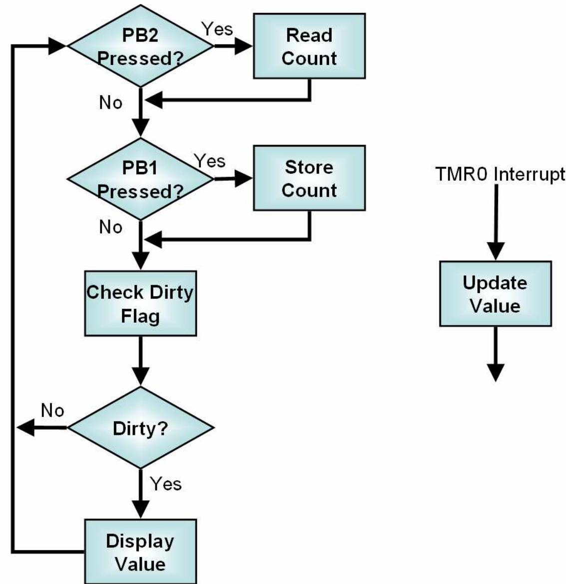
```
        movlw       SAVADR      ; Store the EEPROM address
        banksel     EEADR
        movwf       EEADR       ; to the address register
        banksel     EECON1
        bsf         EECON1,RD   ; again command a read
        banksel     EEDATA
        movf        EEDATA,W    ; grab the high byte
        movwf       binary+1    ; and store in high
```

Continued on next page

Saving the count to EEPROM, Continued

Example B logic

In the second example, the ability to store and restore the count in EEPROM will be added. In this example the EEPROM interrupt is not yet used. It will be used in example C. Here is the logic:



Note that the dirty flag gets set by the “Read Count” block and the “Update Value” block, and cleared by the “Display Value” block.

In the example code the buttons are not debounced. In a real application this should be done to prevent excessive rewriting of the EEPROM.

Building the example code

The project is getting more complex. To build the example code, you must include in your project:

- Source files:
    - ConvBCD2
    - Disp16
    - InitTMR0
    - ISRa
    - L20b
    - RestCnt
    - SaveCntB
- Library files:
    - LCDlib_84A
- Linker Scripts
    - Lesson20

Multiple Interrupt Sources

Introduction

In the previous example, we stored our count in the EEPROM, but we had to spin in a tight loop waiting for the EEPROM write to complete. It would make sense to handle an EEPROM completion interrupt so that our application could be off doing something useful rather than waiting for a flag to change. (Admittedly, the example application is somewhat limited in the "useful" things it has to do).

Managing State

Unfortunately, it isn't quite so simple. In this case, two bytes must be written to EEPROM. The program must write the first byte, wait for an interrupt, write the second byte, and then wait for another interrupt. All this while still processing timer interrupts. Sounds complicated.

Actually, it isn't too bad. A state variable called **eestate** will keep track of what needs to be done next. The variable will be set to zero initially, then to 1 when we need to write the first byte, 2 awaiting completion of the first byte, 3 when the second byte can be written, 4 awaiting completion of that write, and 5 when the operation is done.

All the EEPROM operations, other than those handled by the interrupt service routine, will be consolidated in a subroutine called **SaveCnt**.

eestate	Meaning	Caused by	Next Change
0	No EEPROM activity	SaveCnt	Button Press
1	Write first byte scheduled	main	Write Started
2	Write first byte pending	SaveCnt	Write Complete
3	Write second byte scheduled	ISR	Write Started
4	Write second byte pending	SaveCnt	Write Complete
5	EEPROM write complete	ISR	Ready for next button press

The Button

One detail to handle that wasn't dealt with before. Since the write will be underway while the main program is looping, care must be taken not to initiate a write while one is already in progress. Since **eestate** already keeps track of that detail, this is easily handled:

```
movlw  H'10'        ; Now check whether PB1 pressed
andwf  PORTA,W      ;
btfss  STATUS,Z     ;
goto   CheckDirty   ; No, see if display dirty
movf   eestate,W    ; eestate must be zero to
btfss  STATUS,Z     ; initiate a write
goto   CheckDirty   ; not = 0, don't write
movlw  h'01'        ; Schedule a write because
movwf  eestate      ; we have a value to save
```

Calling SaveCnt

One "trick" that wasn't mentioned in our definition of **eestate**. We have chosen to define it in such a way that our EEPROM handling routine gets called whenever the value of **eestate** is odd:

```
btfsc  eestate,0    ; We call SaveCnt when bit 0
call   SaveCnt      ; is set (eestate = 1, 3, 5)
```

Continued on next page

Multiple Interrupt Sources, Continued

SaveCnt

The SaveCnt routine is called whenever **eestate** is odd. The routine will need to check this value and:

1 – Turn on LED and write first byte
3 – Write second byte
5 – Turn off LED and set **eestate** to 0

We can do this simply with a table:

```
movlw  HIGH(SaveCnt) ; Setup PCLATH for
movwf  PCLATH        ; table call
rrf    eestate,W     ; Need to check only bits
andlw  H'03'         ; 1 and 2
addwf  PCL,F         ; Jump
goto   FirstByte     ; 1 (0)=First byte
goto   NextByte      ; 3 (1)=Second byte
goto   Clearit       ; 5 (2)=Done
```

Note that we know the value is odd since that is the only way we can get here, so we divide **eestate** by two and mask it (for safety) before adding it to the PCL.

The FirstByte and NextByte routines will be the same except for the locations used in the file register and EEPROM. For only two bytes, it isn't worthwhile to try to index these through the FSR, so we will simply duplicate the first part of the code, and then jump to common code:

```
FirstByte
             movlw        H'06'          ; Top LED on
             movwf        PORTB          ; to indicate write in
             movwf        LEDflg         ; progress

             banksel      EECON1         ; Check to be sure a write
             btfsc        EECON1,WR      ; is not already in progress
             return                      ; Yes, we'll be back
             movlw        SAVADR         ; Set address in EEPROM
             banksel      EEADR
             movwf        EEADR          ; to be written
             banksel      binary
             movf         binary,W       ; Set data to be written
             banksel      EEDATA
             movwf        EEDATA         ;
             goto         ContWrite
```

Note that the top LED is turned on by setting a value in **LEDflg**, this is something not done for the second byte. For the second byte, **SAVADR+1** and **binary+1** are used, but otherwise the code is the same. Note that **EECON1** is in bank 1, so a **banksel** is necessary before writing to it. Afterwards, switch back to bank 0. Again, some redundant bank switches are included in the code for clarity.

For both bytes we then jump to:

Continued on next page

Multiple Interrupt Sources, Continued

SaveCnt (continued)

```
ContWrite
              banksel     EECON1       ;
              bcf         INTCON,GIE   ; Turn off interrupts
              bsf         EECON1,WREN  ; Enable write
              movlw       H'55'        ; This sequence is
              movwf       EECON2       ; required before the
              movlw       H'AA'        ; EEPROM may be written
              movwf       EECON2       ;
              bsf         EECON1,WR    ; Start write
              bsf         INTCON,EEIE  ; Enable EEPROM Interrupt
              bsf         INTCON,GIE   ; Re-enable interrupts
```

Note that interrupts are disabled while writing the H'55' and H'AA' to **EECON2**. This sequence must be performed exactly in order for the write to occur. An interrupt during this sequence would cause the write to fail. After writing the 55 and AA as we did in the non-interrupt version, the EEPROM write completion interrupt is enabled:

```
              banksel     eestate
              incf        eestate,F    ; Ready for next step
```

When the write is complete, we want to disable the EEPROM interrupt, reset **eestate** to zero, and turn off the LED:

```
Clearit
              bcf         INTCON,EEIE  ; Disable EEPROM Interrupt
              clrf        eestate      ; Return to state 0
              movlw       H'0e'        ; All LEDs off
              movwf       PORTB
              movwf       LEDflg
```

In the above cases, the various **errorlevel** directives have been omitted for clarity. When they are left out, a number of assembler warnings will be generated. These warnings occur whenever a non-bank 0 location is accessed, whether or not the bank bits have been set correctly.

The Interrupt Service Routine

The interrupt service routine (ISR) must still handle the timer, but it must also deal with the EEPROM interrupt. The interrupt flags must be tested to determine which interrupt caused the ISR to be called:

```
              btfss       INTCON,T0IF  ; Timer interrupt flag
              goto        IRQEEPROM    ; No, go check EEPROM
```

When the EEPROM interrupt is recognized, all that is required is to increment **eestate** (and, of course, clear the interrupt flag):

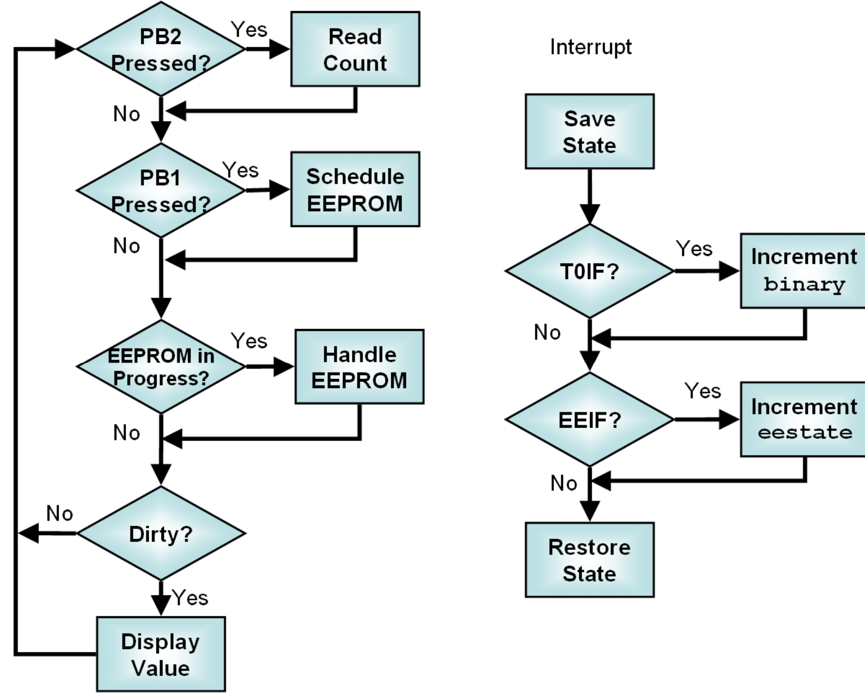
```
              bcf         IFREG,EEIF   ; Clear the interrupt flag
              incf        eestate,F    ; Remember we handled it
```

Continued on next page

Multiple Interrupt Sources, Continued

Overall EEPROM Flow

Handling multiple bytes can get a little confusing. The following figure shows the overall flow of the program logic:



As in the previous example, we set the dirty flag whenever we update the count, and clear the dirty flag whenever we have displayed the count.

Note that the ISR has been kept as simple as possible. Whenever dealing with interrupts, the time that the interrupts are disabled should be kept as short as possible. Since interrupts are disabled while processing an interrupt, this means that the ISR should be as short as possible. Interrupts are also disabled while setting up the EEPROM writes. This was necessary since part of that sequence is critical. However, rather than disabling interrupts for the entire time, we only disabled them for the minimum time necessary. This delay in updating the count is unlikely to be noticed in our application. Note, however, that the TMR0 register continues counting, so while the display update might be put off for a short time, there is no cumulative error in the count.

Continued on next page

Multiple Interrupt Sources, Continued

Testing the Code

The modules that must be included in the project are very similar to the previous exercise, except for the main, the use of the more elaborate ISR, and the more elaborate EEPROM routine:

- Source files:
  - ConvBCD2
  - Disp16
  - InitTMR0
  - ISRc
  - L20c
  - RestCnt
  - SaveCntC
- Library files:
  - LCDlib_84A
- Linker Scripts
  - Lesson20

As before, one should be careful to test all the conditions that might be encountered. Watch to see that the values increment into the second byte, store and restore the count, observe the LED behavior when storing and restoring, and be sure to test restoring a value less than 255 when the count is greater, and a value greater than 255 when the count is less. (Since the value is saved in EEPROM, the stored value should be preserved across reset and power down).

Additional Experiments

Introduction	The PIC-EL does not have inputs on the pins that may be used for other interrupts, so we were unable to run experiments demonstrating the RB0/INT and PORTB change interrupts. However, in Lesson 19, we built a board with a PIC16F87x which has some of those pins free.
RB0/INT	The RB0/INT interrupt allows the PIC to be interrupted when a transition is detected on the RB0 pin (pin 21 on the 872/3/6). This pin is free on the test circuit. The student wishing to test this interrupt might add some circuitry to this pin, perhaps a low speed oscillator such as a 555 to experiment with this interrupt. Note that the pin is triggered either on a rising or falling edge, selectable by the program.
PORTB Change	The PORTB Change interrupt, when enabled, causes an interrupt whenever the input to bits 4 through 7 of PORTB are changed. The experimental board leaves bits 4 and 5 open, and bits 6 and 7 are only used for programming, so they may be used for the experiment provided they are lightly loaded during programming. Using a normally open pushbutton with a high-value pullup would be one way to do this.
Other Interrupts	In addition to the interrupts shared with the PIC16F84A, the PIC16F87x family has a number of other interrupts. There are two additional timers, each of which has an interrupt, there are two capture/compare ports which also have interrupts. These parts contain a serial port which can also notify the program of completion via an interrupt. One of the more interesting interrupts is the A/D completion interrupt. Since we experimented with the A/D in Lesson 19, the interested student might modify the Lesson 19 code to utilize this interrupt.

Wrap Up

Summary	In this lesson we have examined interrupts on the PIC16F84A, and learned how to write to the EEPROM. Both the timer interrupt and the EEPROM write completion interrupt have been used in the examples. The PIC-EL hardware doesn't give us an opportunity to test the 84A's other two interrupt sources, the INT interrupt (transition of RB0) and the PORT B interrupt (change in bits 4-7). Although there are a few things that must be considered, interrupt processing is not terribly difficult, and in some cases, can simplify our code. Interrupts allow writing code that is more responsive to external events, and avoid spending precious compute cycles polling for an event.
Coming Up	In the next lesson, these same 3 exercises will be ported to other processors. Virtually any 18 pin PIC can be used in the PIC-EL, but two of them, the PIC16F54 and the PIC16F716 do not have the on-chip resources to perform the exercises. Both lack EEPROM, and the 54 does not have enough memory. The student wishing to perform the exercises in the next lesson might want to pick up a PIC16F628. The lesson will also examine the 648A, 819 and 88, however those parts are not supported by FPP, so other programming software must be selected. Your author uses DL4YHF's WinPIC for those parts, but there are a large number of choices which can be configured to work with the PIC-EL. The interested student who has installed WinPIC might browse the catalogs of some parts suppliers and pick up other 18 pin PICs that look as if they might be candidates for future projects and can be had at attractive prices.