Chapter

9

INTERRUPTS AND INTERRUPT TIMING

9.1 OVERVIEW

Many microcontroller tasks can be monitored, and paced, by mainline loop timing. Inputs can be sensed for changes every 10 milliseconds. Output changes can be made to occur in response to input changes or in multiples of the 10 millisecond loop time. There are other tasks, however, that require faster handling. For example, transfers taking place at 19,200 baud from the serial port of a computer will present the microcontroller with a new byte of data to be handled every half-millisecond.

To handle this and other tasks requiring a fast response, the PIC18F452 microcontroller contains a wealth of resources. For example, as bits arrive from the serial port of a computer every 50 microseconds or so, the microcontroller's UART (**U**niversal **A**synchronous **R**eceiver **T**ransmitter) proceeds to build the received bits into successive bytes of data. Only as each byte is thus formed does the UART seek the support of the CPU to deal with the received byte. It does so by sending an interrupt signal to the CPU, asking the CPU to suspend what it is doing, deal with the received byte, and then resume the suspended task.

This UART example illustrates two facets of support that the microcontroller gives to fast events. First, built-in modules, operating independently of the CPU, are able to handle a burst of activity before turning to the CPU for help. In addition to the UART, the SPI (**S**erial **P**eripheral **I**nterface) does the same buffering of a byte of data for a shift register interface. The I^2C (**I**nter-Integrated **C**ircuit) interface uses a special protocol to transfer bytes to or from one of several I^2C devices using just two pins of the microcontroller.

Second, a built-in module such as the UART, or one of the timer resources, or even a change on an external interrupt pin can send an interrupt signal to the CPU asking for help. With 17 different interrupt sources, the PIC18F452 provides the designer with a broad range of resources for managing fast events. To give just one example of this breadth, an application requiring a second UART receiver can build one with an external interrupt pin (e.g., RB1/INT1) to signal the start of each new byte and an in-

ternal timer mechanism (e.g., CCP1) to provide an interrupt when each new bit of the received byte is to be read.

This chapter will begin with an examination of the timing issues involved with multiple interrupt sources. Next it will explore how to use a single interrupt source and then multiple interrupt sources. The PIC18F452's two interrupt priority levels will be examined. Working together, they can reduce the *latency* for a high-priority interrupt. The final sections deal with *critical regions* of the mainline code and with the use of external interrupt pins.

9.2 INTERRUPT TIMING FOR LOW-PRIORITY INTERRUPTS

An interrupt source can be characterized by two parameters:

- **TPi**, the time interval between interrupts from interrupt source #i. If this varies from interrupt to interrupt, then **TPi** represents the *shortest* (i.e., the worst-case) interval.
- **Ti**, the time during which the CPU digresses from the execution of the mainline program to handle this one interrupt source. If this varies, then **Ti** represents the *longest* (i.e., the worst-case) value.

For an application with a single interrupt source, the needs of that source will be met if

$$T1 < TP1$$

as illustrated in Figure 9-1. This means that the CPU will complete the execution of the interrupt source's *handler* before it is asked to execute the handler again. The needs of the mainline program will be met if the slices of time left for its execution, TP1 − T1, are sufficient to execute the mainline subroutines during each 10 millisecond loop time. One of the strengths of the PIC18F452's derivation of its 2.5 MHz internal clock rate from a 10 MHz crystal (as on the QwikFlash board) is that both the interrupt source's timing needs and those of the mainline program can be ameliorated *by a factor of 4* simply by changing the programming of a configuration byte so that the chip will run at an internal clock rate of 10 MHz.

In the following discussion, all interrupt sources will be assumed to be fielded with the low-priority interrupt service routine. This is the normal scheme, leaving the PIC18F452's high-priority interrupt service routine available to ameliorate interrupt timing constraints without changing the chip's internal clock rate. Section 9.4 will explore the help provided by high-priority interrupts.

The worst-case timing diagrams for *two* interrupt sources are illustrated in Figure 9-2. The first is the worst-case timing diagram for interrupt source #1 (IS#1). Just before it requests service, IS#2 requests and gets service. Because the CPU's servicing of IS#2 automatically disables, temporarily, all other interrupts, IS#1 is put off until the handler for IS#2 has run to completion. This time is labeled T2, the duration of the handler. Thus the *worst-case latency* for IS#1 is this same value, T2. It can be seen that even in this worst case for IS#1, it is serviced well before it requests service again. In general, the condition IS#1 must meet is

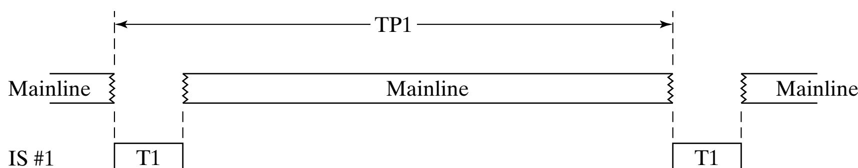


Figure 9-1 Interrupt timing parameters.

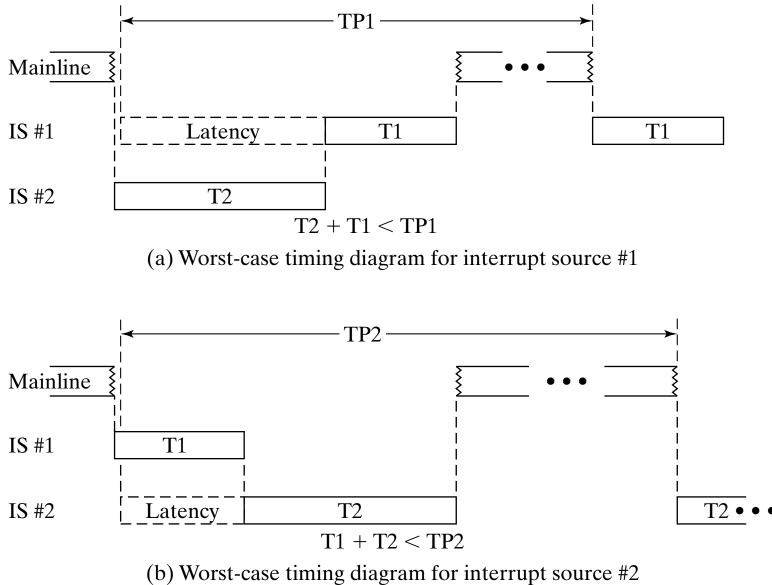


Figure 9-2 Worst-case timing diagrams for two interrupt sources.

$$T2 + T1 < TP1$$

Figure 9-2b illustrates the worst-case timing diagram for IS#2. In this case, IS#2 requests service just after IS#1 has requested, and received, service. Again, so long as IS#2 receives service before it requests service again, it satisfies its timing requirements. In this case of two interrupt sources, the requirement can be expressed

$$T1 + T2 < TP2$$

For three or more interrupt sources, the test that each interrupt source must satisfy depends on the assignment of its handler in the interrupt service routine's polling routine listed in Figure 6-2 and repeated in Figure 9-3. Reassigning the order in which interrupt sources are polled can sometimes rectify an assignment that produces a worst-case timing problem. Two factors account for the worst-case timing constraint for each interrupt source:

1. In the worst case, an interrupt source will ask for service *just after* the longest handler further down the polling routine has begun.
2. In addition, all handlers above it will also be serviced first.

The first condition occurs because once the execution of *any* handler has begun, further interrupts are automatically disabled. Therefore, the execution of the handler (regardless of where it ranks in the polling routine) will play out to completion. The second condition is a result of the

```
CONTINUE_
```

construct being executed after each handler. This construct causes execution to revert to the beginning of the polling routine where, in the worst case, all interrupt sources above the source in question will be waiting for service.

```
LOOP_

  IF_  ‹test whether interrupt #1 is ready for service›
    rcall  Int1handler
    CONTINUE_
  ENDIF_

  IF_  ‹test whether interrupt #2 is ready for service›
    rcall  Int2handler
    CONTINUE_
  ENDIF_

  IF_  ‹test whether interrupt #3 is ready for service›
    rcall  Int3handler
    CONTINUE_
  ENDIF_
   .
   .
   .
  IF_  ‹test whether interrupt #N is ready for service›
    rcall  IntNhandler
    CONTINUE_
  ENDIF_

  BREAK_

ENDLOOP_
```

Figure 9-3 Interrupt service routine's polling routine.

These considerations lead to the following interrupt timing constraints for four interrupt sources:

$$\begin{aligned} (\text{maximum of T2,T3,T4}) + T1 &< TP1 \\ (\text{maximum of T3,T4}) + T1 + T2 &< TP2 \\ T4 + T1 + T2 + T3 &< TP3 \\ T1 + T2 + T3 + T4 &< TP4 \end{aligned} \quad \textbf{(9-1)}$$

The low-priority interrupt service routine is shown in its entirety in Figure 9-4. When an interrupt occurs (with interrupts enabled to the CPU), the following sequence of events takes place automatically:

- The CPU completes the execution of its present mainline instruction.
- The low-priority global interrupt enable bit (**GIEL**) is cleared, thereby disabling further low-priority interrupts.
- The contents of the program counter (containing the address of the next mainline program instruction to be executed upon return from the interrupt service routine) is stacked.
- The program counter is loaded with 0x0018, the low-priority interrupt vector.

In order not to cause erroneous operation of the mainline code it is interrupting, the interrupt service routine must be sure to return CPU registers the way they were found. Because it is difficult to do *anything* without changing the contents of **WREG** and the **STATUS** register, these are set aside at the beginning of the interrupt service routine and restored at the end. Note the order of restoration: first **WREG** and then **STATUS**. This order matters because the

```
movf  WREG_TEMP,W
```

instruction affects the **Z** and the **N** bits of the **STATUS** register. By restoring **STATUS** last, the mainline code will get these bits back *exactly* as they were left by the last instructions executed in the mainline code before the interrupt occurred.

The assumptions listed in Figure 9-4c are suggested simply to reduce the number of *other* CPU registers that must be set aside and restored. As suggested in Section 2-3, the **BSR** register can be set to 0x01

```
;;;;;;; Vectors ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        org  0x0000             ;Reset vector
        nop
        goto  Mainline

        org  0x0008             ;High-priority interrupt vector address
        goto  $                 ;Trap

        org  0x0018             ;Low-priority interrupt vector address
        goto  LoPriISR
```

(a) Vectors.

```
LoPriISR                        ;Low-priority interrupt service routine
        movff  STATUS,STATUS_TEMP  ;Set aside STATUS and WREG
        movwf  WREG_TEMP

        LOOP_
          IF_  <test whether interrupt #1 is ready for service>
            rcall  Int1handler
             CONTINUE_
          ENDIF_

          IF_  <test whether interrupt #2 is ready for service>
            rcall  Int2handler
            CONTINUE_
          ENDIF_

          IF_  <test whether interrupt #3 is ready for service>
            rcall  Int3handler
            CONTINUE_
          ENDIF_
           .
           .
          IF_  <test whether interrupt #N is ready for service>
            rcall  IntNhandler
            CONTINUE_
          ENDIF_

          BREAK_
        ENDLOOP_

        movf  WREG_TEMP,W       ;Restore WREG and STATUS
        movff  STATUS_TEMP,STATUS
        retfie                  ;Return from interrupt, reenabling GIEL
```

(b) Low-priority interrupt service routine.

BSR is never changed throughout the application.
FSR0 and **FSR1** are used only in the mainline code.
FSR2 is used only in interrupt handlers.
PCL is never used as an operand in an interrupt handler.

(c) Assumptions.

Figure 9-4 Low-priority interrupt mechanism.

in the **Initial** subroutine and thereafter never changed. This will make 128 + 256 = 384 bytes of RAM reachable by direct addressing, an adequate number for most application programs. The remaining RAM is still reachable, but by means of indirect addressing with **FSR0**, **FSR1**, or **FSR2**.

It is quite common to use indirect addressing within an interrupt handler. For example, successive characters received by the UART might be written into a *line buffer* with the

```
    movwf  POSTINC2
```

instruction. If the interrupt handlers all avoid using **FSR0** and **FSR1**, and if the mainline code avoids using **FSR2**, then none of these 2 byte registers need be set aside and restored.

The last assumption of Figure 9-4c of never using PCL as an operand in an interrupt handler is listed as a reminder that such an operation may change the content of **PCLATH**. If it is desired to build a *jump table* (see Problem 2-9) into an interrupt handler, it is only necessary to set aside and restore **PCLATH**.

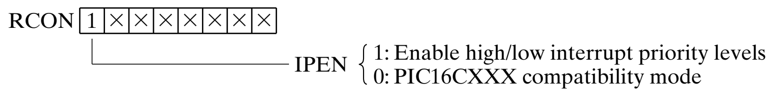
The interrupt timing constraints for four interrupt sources listed earlier as (9-1) correctly identify items that must be considered in each case. However, in the interest of keeping the explanation simple, some small items were left out. For example, the polling routine of Figure 9-3 adds a few cycles as each test is carried out and the associated branch instruction executed. Likewise, the automatic vectoring from mainline code to interrupt service routine inserts a couple of cycles, as does the setting aside and restoring of **WREG** and **STATUS**. Nevertheless, the timing constraints of (9-1) keep the focus on the dominant factors that a designer can do something about if the timing is close.

9.3 LOW-PRIORITY INTERRUPT STRUCTURE

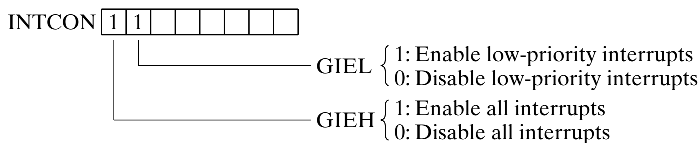
Using the high-priority/low-priority interrupt scheme built into the PIC18F452 begins with the setting of the **IPEN** bit shown in Figure 9-5a with

```
bsf  RCON,IPEN           ;Enable priority levels
```

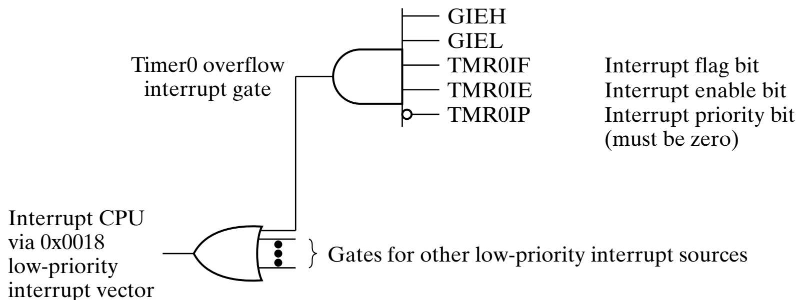
The alternative, **IPEN** = 0, causes the chip to revert to the single interrupt level scheme of earlier-generation PIC microcontrollers, discarding a valuable feature of this chip.



(a) Initialization for two levels of interrupt priority



(b) Global interrupt enable bits



(c) Structure for low-priority interrupt sources.

Figure 9-5 Low-priority interrupt structure.

The **GIEH** (**G**lobal **I**nterrupt **E**nable for **H**igh-priority interrupts) bit gates *all* interrupts to the CPU, both high priority and low priority. It is usually set by the last instruction before the **return** from the **Initial** subroutine, after each interrupt source being used in an application has been initialized.

The **GIEL** (**G**lobal **I**nterrupt **E**nable for **L**ow-priority interrupts) bit gates all low-priority interrupts to the CPU. It, too, is set by one of the last instructions in the **Initial** subroutine. It is automatically cleared when a low-priority interrupt occurs, blocking further automatic vectoring if a second low-priority interrupt occurs while a first one is being serviced. The **GIEL** bit is automatically set again by the execution of the

```
retfie                    ;Return from interrupt
```

instruction at the close of the interrupt service routine.

The **GIEL** bit can also be used within the mainline code to disable low-priority interrupts while a *critical region* of code extending over a handful of instructions is executed, followed by the reenabling of interrupts by setting the **GIEL** bit again. One occasion that called for such treatment arose in conjunction with the **LoopTime** subroutine of Section 5-4.

Each interrupt source has associated with it an interrupt priority bit that assigns the interrupt source to either the high-priority interrupt structure (discussed in Section 9.4) or the low-priority interrupt structure discussed here. The default state of these interrupt-priority bits at power-on reset assigns every interrupt source to the high-priority interrupt structure. Accordingly, the "IP" bit for each interrupt source to be assigned to the low-priority interrupt structure must be *cleared* in the **Initial** subroutine. For example,

```
bcf  INTCON2,TMRØIP
```

will assign Timer0 overflow interrupts to the low-priority interrupt structure.

Because the default state of the "IP" bit must be changed under the normal circumstance of assigning an interrupt source to the low-priority interrupt structure, it becomes necessary to know where each of these bits is located. Figure 9-6 lists them all, along with each interrupt source's local enable bit and its flag bit.

Example 9-1 The **LoopTime** subroutine discussed in Section 5.4 was able to use the setting of the **TMR0IF** flag to obtain precise timing for a 10 millisecond loop time with an internal clock rate of 2.5 MHz and the use of Timer0 as a scale-of-25,000 counter. Some precision was lost in trying to achieve a 10 millisecond loop time with an internal clock rate of 10 MHz and the use of Timer0 as a scale-of-100,000 counter. The counter required use of Timer0's prescaler, and the write to the timer reset the prescaler. For most applications, the resulting error is miniscule. With the help of Timer0 overflow interrupts, the error can be eliminated. Develop the interrupt routine and the modified **LoopTime** subroutine.

Solution

A **TMR0handler** interrupt handler can be made to set the **TMR0IF** flag precisely every 50,000 cycles, or 5 microseconds. Each time it does so, it decrements a **TMR0CNT** variable. The **LoopTime** subroutine now waits for **TMR0CNT** to be equal to zero as its signal that 10 milliseconds have elapsed. It then simply reinitializes **TMR0CNT** to 2. The resulting subroutine is listed in Figure 9-7a. The initialization for Timer0 interrupts is shown in Figure 9-7b. The **TMR0handler** is listed in Figure 9-7c. The **LoPriISR** interrupt service routine is shown in Figure 9-7d, assuming there are no other interrupt flags to poll.

Name	Priority Bit	Local Enable Bit	Local Flag Bit
INT0 external interrupt	*	INTCON,INT0IE	INTCON,INT0IF
INT1 external interrupt	INTCON3,INT1IP	INTCON3,INT1IE	INTCON3,INT1IF
INT2 external interrupt	INTCON3,INT2IP	INTCON3,INT2IE	INTCON3,INT2IF
RB port change interrupt	INTCON2,RBIP	INTCON,RBIE	INTCON,RBIF
TMR0 overflow interrupt	INTCON2,TMR0IP	INTCON,TMR0IE	INTCON,TMR0IF
TMR1 overflow interrupt	IPR1,TMR1IP	PIE1,TMR1IE	PIR1,TMR1IF
TMR3 overflow interrupt	IPR2,TMR3IP	PIE2,TMR3IE	PIR2,TMR3IF
TMR2 to match PR2 int.	IPR1,TMR2IP	PIE1,TMR2IE	PIR1,TMR2IF
CCP1 interrupt	IPR1,CCP1IP	PIE1,CCP1IE	PIR1,CCP1IF
CCP2 interrupt	IPR2,CCP2IP	PIE2,CCP2IE	PIR2,CCP2IF
A/D converter interrupt	IPR1,ADIP	PIE1,ADIE	PIR1,ADIF
USART receive interrupt	IPR1,RCIP	PIE1,RCIE	PIR1,RCIF
USART transmit interrupt	IPR1,TXIP	PIE1,TXIE	PIR1,TXIF
Sync. serial port int.	IPR1,SSPIP	PIE1,SSPIE	PIR1,SSPIF
Parallel slave port int.	IPR1,PSPIP	PIE1,PSPIE	PIR1,PSPIF
Low-voltage detect int.	IPR2,LVDIP	PIE2,LVDIE	PIR2,LVDIF
Bus-collision interrupt	IPR2,BCLIP	PIE2,BCLIE	PIR2,BCLIF

* INT0 can only be used as a high-priority interrupt

Figure 9-6 Register and bit names for every interrupt source.

Example 9-2 Determine the "T1" and "TP1" values for the Timer0 interrupts of the last example. Also determine the percentage of the CPU's time spent handling these interrupts.

Solution

The interval between interrupts, TP1, is 50,000 cycles or 5,000 microseconds, given the 10 MHz internal clock rate of the chip. When a Timer0 overflow interrupt occurs, the CPU takes two cycles after executing the last mainline instruction before it executes the interrupt vector instruction

```
goto  LoPriISR
```

Consequently,

$$T1 = 2 + 12 + 18 = 32 \text{ cycles}$$

The percentage of the CPU's time spent handling Timer0 interrupts is

$$(32/50000) \times 100 = 0.064\%$$

Example 9-3 If another low-priority interrupt were added to the Timer0 interrupts of the last problem, and if the Timer0 interrupts were placed second in the polling routine, what would be the new interrupt source's worst-case latency because of the Timer0 interrupts?

```
LoopTime
        REPEAT_                 ;Wait until interrupt decrements TMRØCNT to zero
          movf  TMRØCNT,F
        UNTIL_  .Z.
        MOVLF  2,TMRØCNT
        return
```

(a) **LoopTime** subroutine.

```
        bsf  RCON,IPEN          ;Enable two interrupt priority levels
        bcf  INTCON2,TMRØIP     ;Assign TMRØ low interrupt priority
        bcf  INTCON,TMRØIF      ;Clear TMRØ overflow flag
        bsf  INTCON,TMRØIE      ;Enable TMRØ overflow interrupt source
        MOVLF  2,TMRØCNT        ;Initialize counter
        bsf  INTCON,GIEL        ;Enable low-priority interrupts to CPU
        bsf  INTCON,GIEH        ;Enable all interrupts to CPU
```

(b) Instructions to be added to the **Initial** subroutine.

```
Bignum  equ  65536-5ØØØØ+12+2
TMRØhandler
        decf  TMRØCNT,F         ;Decrement counter                            (1)
        bcf  INTCON,GIEH        ;Disable interrupts                           (1)
        movff  TMRØL,TMRØLCOPY  ;Read 16-bit counter at this moment           (2)
        movff  TMRØH,TMRØHCOPY  ;                                             (2)
        movlw  low  Bignum      ;                                             (1)
        addwf  TMRØLCOPY,F      ;                                             (1)
        movlw  high  Bignum     ;                                             (1)
        addwfc  TMRØHCOPY,F     ;                                             (1)
        movff  TMRØHCOPY,TMROH  ;                                             (2)
        movff  TMRØLCOPY,TMROL  ;Write 16-bit counter at this moment          (2)
        bsf  INTCON,GIEH        ;Reenable interrupts                          (1)
        bcf  INTCON,TMRØIF      ;Clear TimerØ flag                            (1)
        return                  ;                                             (2)
```

(c) **TMR0handler** subroutine (18 cycles).

```
        org  Øx0Ø18             ;Low-priority interrupt vector address
        goto  LoPriISR          ;Jump                                         (2)
          .
          .
LoPriISR                        ;Low-priority interrupt service routine
        movff  STATUS,STATUS_TEMP  ;                                          (2)
        movwf  WREG_TEMP        ;                                             (1)
        rcall  TMRØhandler      ;                                             (2)
        movf  WREG_TEMP,W       ;                                             (1)
        movff  STATUS_TEMP,STATUS  ;                                          (2)
        retfie                  ;                                             (2)
```

(d) **LoPriISR** routine (12 cycles).

Figure 9-7 Example 9-1.

Solution

In the worst case, a Timer0 interrupt would have occurred, **STATUS** and **WREG** set aside, and the polling routine would have found the new interrupt source *not* asking for service. At that precise moment (in the worst case), the new interrupt source would set its flag, asking for service. Meanwhile, in **LoPriISR**, where interrupts are disabled, the CPU would execute

- The branch associated with the **IF_** construct for the new interrupt (2 cycles)
- The test of the **TMR0IF** flag (2 cycles)

- The call of **TMR0handler** (2 cycles)
- The handler itself (18 cycles)
- The branch for the **CONTINUE_** construct following the return to the polling routine (2 cycles)
- The test of the new interrupt's flag (2 cycles)
- The call of the new interrupt's handler (2 cycles)

After this worst-case latency of 30 cycles, or 3 microseconds, the CPU would execute the first instruction of the new interrupt's handler.

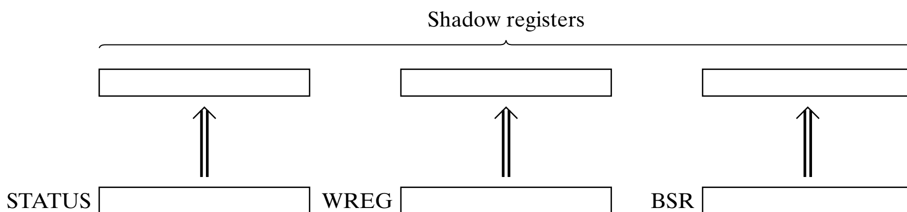
9.4 HIGH-PRIORITY INTERRUPT STRUCTURE

An interrupt source assigned with its "IP," interrupt priority, bit to the high-priority interrupt structure gains the benefit of being able to suspend the execution of the mainline code and to disable all low-priority interrupts. Furthermore, it can even suspend the execution of the low-priority interrupt service routine, **LoPriISR**. Except for any brief disabling of high-priority interrupts to protect a critical region of code, a *single* interrupt source assigned to the high-priority interrupt structure experiences *no* latency at all! This benefit quickly dissipates as soon as a second interrupt source is also assigned high priority.

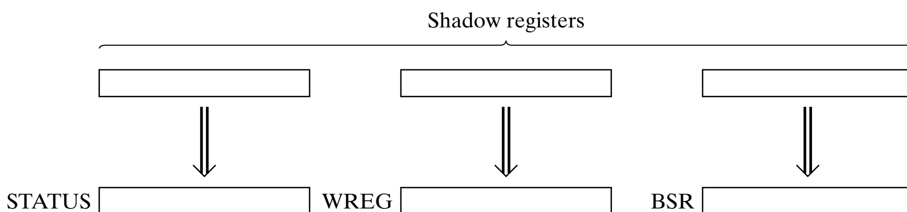
The designers of the PIC18F452 added one further feature to minimize the latency of a high-priority interrupt. As shown in Figure 9-8a, when a high-priority interrupt occurs, the contents of **STATUS**, **WREG**, and **BSR** are automatically copied to *shadow registers.* Once the interrupt source has been serviced, the execution of

```
retfie  FAST
```

tells the CPU to automatically



(a) Automatic setting aside of **STATUS**, **WREG**, and **BSR** when a high-priority interrupt occurs



(b) Automatic restoration of **STATUS**, **WREG**, and **BSR** in response to the "**retfie FAST**" instruction.

Figure 9-8 Use of high-priority interrupt's shadow registers.

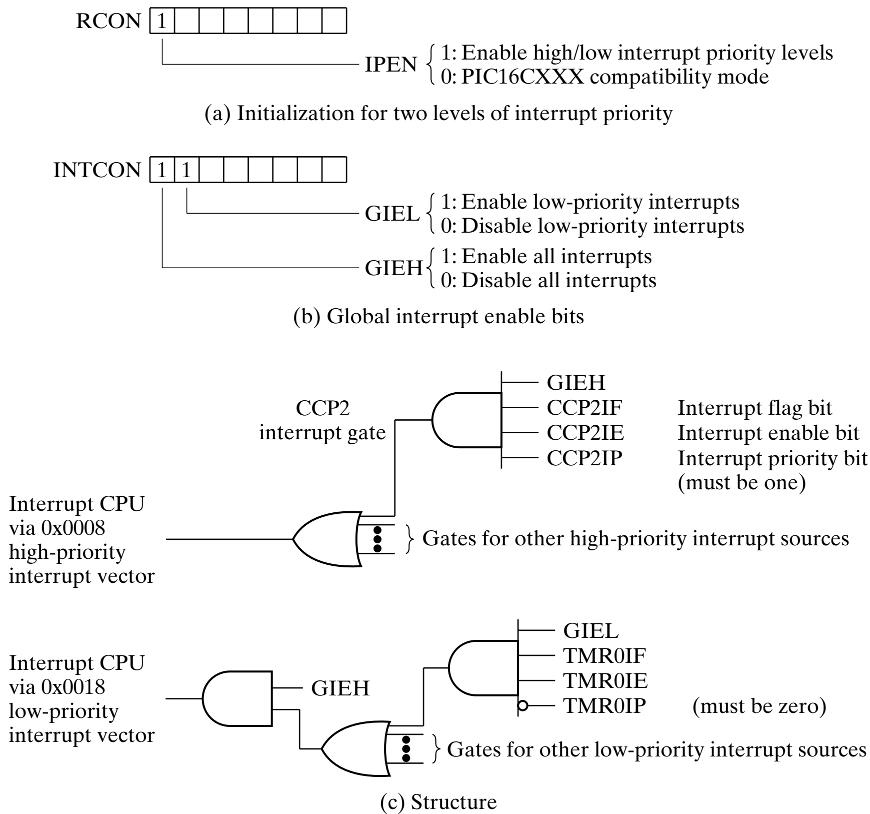


Figure 9-9 High-priority/low-priority interrupt structure.

1. Restore not only the program counter, but also **STATUS**, **WREG**, and **BSR**
2. Restore the **GIEH** bit, reenabling both high- and low-priority interrupts, as shown in Figure 9-9

Example 9-4 In Section 13.9, it will be seen that the frequency of a square wave can be measured with the 50 parts-per-million accuracy of the microcontroller's crystal oscillator. The PIC18F452's "CCP2" input will be used to generate a high-priority interrupt for every 16th rising edge of the input waveform. These interrupts will be counted over an interval of about 1 second. Knowing the exact number of internal clock cycles (e.g., 2500540) over which an integral number of periods of the input waveform takes place (e.g., $16 \times 123456 = 1975296$) gives the information needed to calculate the input frequency. If the microcontroller's internal clock period is 0.4 microseconds, the frequency is given by

$$\text{Frequency} = \frac{1975296}{2500540 \times 0.4} = 1.97487\ \text{MHz}$$

The time required to count every 16th input edge will determine the maximum frequency that can be measured. Show the high-priority interrupt service routine to increment a 3 byte counter,

FCOUNTU:FCOUNTH:FCOUNTL

```
                org  0x0008              ;High-priority interrupt vector address
                goto  HiPriISR           ;                                        (2)
                 .
                 .
                 .
        HiPriISR                         ;High-priority interrupt service routine
                bcf  PIR2,CCP2IF         ;Clear interrupt flag                    (1)
                clr  WREG                ;Clear WREG for subsequent adds with carry  (1)
                incf  FCOUNTL,F          ;Add 1 to three-byte value in FCOUNT     (1)
                addwfc  FCOUNTH,F        ;                                        (1)
                addwfc  FCOUNTU,F        ;                                        (1)
                retfie  FAST             ;Return and restore from shadow registers   (2)
```

Figure 9-10 Example 9-4.

Solution

The high-priority interrupt service routine is shown in Figure 9-10. With two cycles to get from the execution of interrupted code to execution of the

```
goto  HiPriISR
```

two more cycles for this **goto** instruction, and seven cycles for the **HiPriISR**, the CPU digresses from the interrupted code for eleven cycles, or

$$11 \times 0.4 = 4.4 \text{ microseconds}$$

The maximum frequency that can be measured is

$$\frac{16}{4.4} = 3.6 \text{ MHz}$$

9.5 CRITICAL REGIONS

A *critical region* of code is a sequence of program instructions that *must not* be interrupted if erroneous operation is to be avoided. An example arose in the **LoopTime** subroutine. Timer0 was read, manipulated, and rewritten. Correct operation required that exactly 12 cycles occurred between the read and the rewrite. An intervening interrupt would have thrown off this count, causing an extension of the loop time.

A resource accessed by both the mainline code and an interrupt handler may have the potential for a malfunction.

Example 9-5 Consider the three-LED array of the QwikFlash board driven from **PORTA**, as shown in Figure 4-2a. An interrupt routine is to set **RA3** when a rarely occurring condition occurs. If the LED is on, the user knows that the condition has occurred. Meanwhile, suppose that **RA2** and **RA1** are used by the mainline code to echo the state of the RPG (**PORTD**'s **RD1** and **RD0**) to give a visual indication of RPG changes. Show the code to echo the RPG state on the LEDs, describe the possible malfunction, and provide a solution.

Solution

One solution to echoing the RPG output to the two LEDs is shown in Figure 9-11a. **PORTA** is copied to **WREG**, the two bits that will hold the RPG bits are cleared to zero, and the result saved to **TEMP**. Next **PORTD** is copied to **WREG**, shifting bits 1 and 0 of **PORTD** to bits 2

```
movf  PORTA,W           ;Read PORTA and mask off bits 2 and 1
andlw  B'11111001'
movwf  TEMP             ;and save the result in TEMP
rlncf  PORTD,W          ;Shift PORTD one place left and into WREG
andlw  B'00000110'      ;Mask off all but bits 2 and 1
iorwf  TEMP,W           ;OR TEMP into this
movwf  PORTA            ;and return it to PORTA
```

(a) Original code with critical region problem.

```
bcf  <register>,<bit>   ;Disable the local interrupt enable bit
movf  PORTA,W           ;Read PORTA and mask off bits 2 and 1
andlw  B'11111001'
movwf  TEMP             ;and save the result in TEMP
rlncf  PORTD,W          ;Shift PORTD one place left and into WREG
andlw  B'00000110'      ;Mask off all but bits 2 and 1
iorwf  TEMP,W           ;OR TEMP into this
movwf  PORTA            ;and return it to PORTA
bsf  <register>,<bit>   ;Reenable local interrupt enable bit
```

(b) Solution by disabling the interrupt source that changes RA3.

```
movlw  B'11111001'      ;Force RA2 and RA1 to zero
andwf  PORTA,F
rlncf  PORTD,W          ;Move RD1 and RD0 to bits 2 and 1 of WREG
andlw  B'00000110'      ;Force all other bits to zero
iorwf  PORTA,F          ;and OR this back into PORTA
```

(c) Solution by changing **PORTA** with read-modify-write instructions.

```
IF_  PORTD,RD1 == 1     ;Copy RD1 to RA2
  bsf  PORTA,RA2
ELSE_
  bcf  PORTA,RA2
ENDIF_
IF_  PORTD,RD0 == 1     ;Copy RD0 to RA1
  bsf  PORTA,RA1
ELSE_
  bcf  PORTA,RA1
ENDIF_
```

(d) Alternative solution changing **PORTA** with read-modify-write instructions.

Figure 9-11 Example 9-5.

and 1 of **WREG**. The remaining bits of **WREG** are forced to zero, the result is ORed with the manipulated copy of **PORTA** located in **TEMP**, and the result returned to **PORTA**.

Note that if the interrupt occurs and sets bit 3 of **PORTA** anytime after the read of **PORTA** and before the write back to **PORTA**, then bit 3 of **PORTA** will be cleared back to its original state by the write back to **PORTA**.

The chance of the interrupt occurring at the precise moment this mainline sequence is being executed is remote. Consequently, the resulting code bug is difficult to find. Better solutions exist that absolutely avoid the problem. Figure 9-11b treats the mainline code as a critical region and postpones for just a few microseconds the execution of the specific interrupt handler that deals with the rarely occurring condition. An even better solution is to access **PORTA** with nothing but the microcontroller's read-modify-write instructions. The problem in Figure 9-11a arose because an interrupt could intervene between the initial read of **PORTA** and the final write to **PORTA**. In the code of Figure 9-11c the

```
andwf  PORTA,F
```

reads **PORTA**, modifies it, and writes the result back to **PORTA**, all in one instruction. Because an interrupt will not break into the middle of an instruction, the integrity of the read-modify-

write sequence is not compromised. A little later in the sequence of Figure 9-11c, **PORTA** is again subjected to a read-modify-write instruction with the same result.

A third solution is shown in Figure 9-11d. In this case, only the read-modify-write

```
bsf
```

and

```
bcf
```

instructions are used to change **PORTA**, with the same error-free result.

A fourth solution would have the interrupt service routine set one bit of a flag variable (rather than RA3 directly). Then each time around the mainline loop, the CPU can check the flag bit. If it is set, then the CPU sets RA3.

9.6 EXTERNAL INTERRUPTS

The PIC18F452 has three external interrupt inputs:

INT0 INT1 INT2

These are shared with bits 0, 1, and 2 of **PORTB**. To use one of these as an interrupt source, its control bits must be set up, using the information of Figure 9-12.

Example 9-6 Set up INT1 as a falling-edge-sensitive interrupt input having low priority.

Solution

The following code will suffice:

```
bsf  TRISB,1             ;Input
bcf  INTCON2,INTEDG1     ;Falling-edge sensitive
bcf  INTCON3,INT1IP      ;Low priority
bcf  INTCON3,INT1IF      ;Clear flag
bsf  INTCON3,INT1IE      ;Enable interrupt source
bsf  INTCON,GIEL         ;Enable low-priority interrupts
bsf  INTCON,GIEH         ;Enable all interrupts
```

When a falling edge occurs on the INT1 input, the CPU will set aside what it is doing and vector through the low-priority interrupt vector at 0x0018 to the low-priority interrupt service routine, as described in Figure 9-4. Within the **INT1handler** subroutine, the interrupt flag can be cleared with

```
bcf  INTCON3,INT1IF
```

along with the code whose execution has been triggered by the falling edge on the INT1 input pin.

Example 9-7 Use the INT1 pin to generate a low-priority interrupt on both falling and rising edges.

Solution

Within the INT1 handler include

```
btg  INTCON2,INTEDG1     ;Toggle edge sensitivity
```

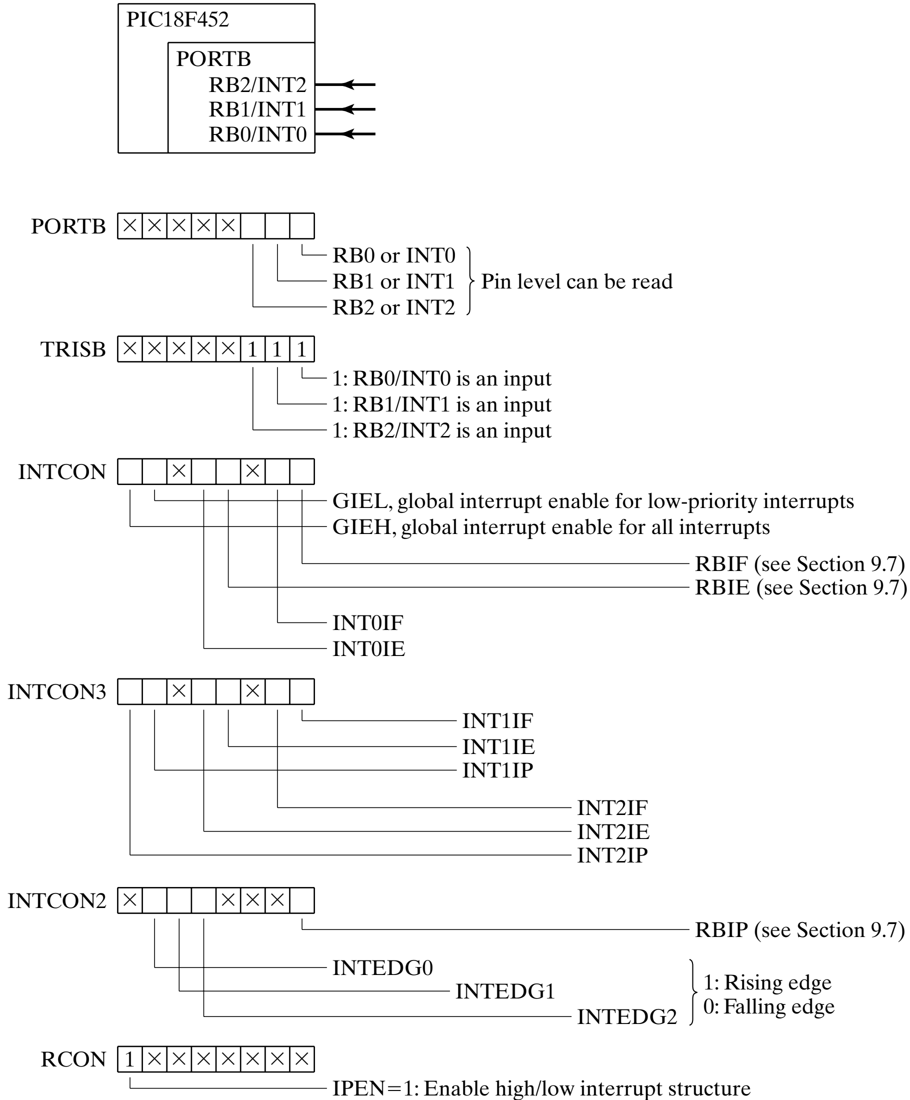


Figure 9-12 External interrupts.

9.7 PORTB-CHANGE INTERRUPTS (PINS RB7:RB4)

A low-to-high change or a high-to-low change on any of the upper four pins of **PORTB** *that are set up as inputs* can be used to generate an interrupt. Circuitry associated with **PORTB** keeps a copy of the state of these four pins as they were when the port was last read from or written to. Any subsequent mismatch caused by the change of an *input* pin among bits 7, 6, 5, 4 of **PORTB** will set the **RBIF** (register B interrupt flag) bit in the **INTCON** register. If interrupts have been set up appropriately (with **RBIE**,

RBIP, **GIEL**, and **GIEH**), then the CPU will be interrupted. An interrupt handler will respond to the **PORTB** change. The **RBIF** flag is cleared by a two-step process:

1. Read **PORTB** (or write to it) to copy the upper four bits of **PORTB** into the hardware copy, thereby removing the mismatch condition.
2. Execute

```
bcf  INTCON,RBIF
```

Note that once the **RBIF** flag has been set, the first step in clearing it may be carried out unintentionally if some unrelated routine accesses **PORTB**. For example,

```
bsf  PORTB,2
```

will carry out the first step needed to clear the **RBIF** bit. However, until the second step

```
bcf  INTCON,RBIF
```

is carried out, the flag will remain set. Consequently, a polling routine will work correctly in spite of reads and writes of **PORTB** by unrelated code.

A problem can arise if one of the inputs among the upper 4 bits of **PORTB** should happen to change at *the exact moment* that **PORTB** is being accessed by unrelated code. In this rarely occurring case, **RBIF** may not get set. This problem is a potential source of system malfunction any time **PORTB**-change interrupts are used. A better use of this facility arises when the microcontroller is put into its power-saving sleep mode. All code execution stops. A change on one of the **PORTB** upper pins can be used to awaken the microcontroller. Because there will never be a conflict between this occurrence and the execution of an instruction accessing **PORTB** (because code execution is stopped), the change on the **PORTB** pin will *never go* unnoticed.

PROBLEMS

9-1 Polling sequence. An application requires three interrupt sources having the following characterizing times:

TA = 10 μs   TPA = 2500 μs
TB = 10 μs   TPB = 250 μs
TC = 10 μs   TBC = 25 μs

For simplicity, assume that these times are the only times arising in the interrupt service routine (e.g., that all the extra tests and branches of the polling routine take no time).

(a) Show the worst-case timing diagram for IS#C if the interrupts are assigned to the polling routine in the order A, B, C. Will IS#C be serviced properly under all circumstances?
(b) Repeat with the polling routine order C, B, A.

9-2 Worst-case interrupt timing constraints. Consider the interrupt timing constraints for the four interrupt sources labeled (9-1) in Section 9-2.

(a) Why does the second constraint, imposed by IS#2, depend on the maximum of T3 and T4?
(b) Why does the constraint imposed by IS#3 equal that imposed by IS#4 if TP3 = TP4? That is, why does the higher position in the polling sequence not help IS#3?

9-3 LoPriISR. In the discussion of Figure 9-4, it was mentioned that the *order* of restoration of **WREG** and **STATUS** at the end of the interrupt service routine matters. Does the order of setting aside **WREG** and **STATUS** at the beginning of the interrupt service routine matter? Explain.

9-4 LoPriISR assumptions. Consider the assumptions of Figure 9-4c. If several interrupt handlers need to use indirect addressing, then **FSR2** (consisting of the two bytes, **FSR2H** and **FSR2L**) must be shared between them.

(a) Show the code used by IS#1 at the beginning of its handler to load its own pointer from **FSR21H:FSR21L** into **FSR2**.
(b) Show the code used by IS#1 at the end of its handler to save the content of **FSR2** back in **FSR21H:FSR21L**.
(c) How may cycles does this juggling of the content of **FSR2** add to the handler for IS#1?

9-5 LoopTime subroutine for 10 MHz operation. The **LoopTime** subroutine of Figure 9-7a causes **TMR0CNT** to count . . . , 1, 0 → 2, 1, 0 → 2, 1, 0 → 2, . . . in a two-state sequence.

(a) If the rest of the mainline code takes only 2 milliseconds to execute during a given pass around the mainline loop, what will be the state of **TMR0CNT** when the **LoopTime** subroutine is entered?
(b) Answer part (a) if the rest of the mainline code takes 7 milliseconds to execute.
(c) Answer part (a) if, on rare occasions, the rest of the mainline code takes 12 milliseconds to execute. What will be the effect of this on the performance of the **LoopTime** subroutine?
(d) Rewrite the **LoopTime** subroutine to test the most-significant bit (MSb) of **TMR0CNT**. When this bit becomes set, as **TMR0CNT** decrements from 0x00 to 0xff, increment **TMR0CNT** twice and then return from the subroutine.
(e) Given this change in the **LoopTime** subroutine, reanswer parts (a), (b), and (c).

9-6 Worst-case latency. Example 9-3 asked for the worst-case latency experienced by a second low-priority interrupt due to the Timer0 interrupts of Figure 9-7. This ignored the effect of a high-priority interrupt service routine. If, in fact, the high-priority interrupt service routine of Figure 9-10 were also employed in the application, then what would be the worst-case latency experienced by the second low-priority interrupt? Show a worst-case timing diagram.

9-7 Minimum latency. The text at the beginning of Section 9.4 implied that zero latency could be attained for an interrupt source if the application code never needed to disable high-priority interrupts to protect a critical region of code and if the "zero latency" interrupt source were made a high-priority interrupt. Actually, the **HiPriISR** code of Figure 9-10 exhibits a nonzero fixed latency, from the time the interrupt's flag is set until it is executing the first instruction of **HiPriISR**. Three cycles are automatically inserted by the CPU between the flag setting at the end of one cycle and the execution of a 1 byte instruction at vector address 0x0008. Four cycles are automatically inserted between the flag setting and the execution of a 2 byte "goto" instruction. Thus, the first (1 byte) instruction of **HiPriISR** is executed on the fourth or fifth cycle after the flag is set.

(a) What will be the latency if the

```
goto  HiPriISR
```

of Figure 9-10 is replaced by the **HiPriISR** interrupt service routine itself?

(b) In general, how many instruction words can **HiPriISR** contain before it impinges on the low-priority ISR vector?

9-8 Shadow registers. The shadow register mechanism of Figure 9-8 can be used by a high-priority interrupt simply by terminating **HiPriISR** with the instruction

```
retfie  FAST
```

Assuming that **LoPriISR** is terminated (as it should be) with

```
retfie
```

would it matter whether the shadow register mechanism actually worked as in Figure 9-8a for low-priority interrupts as well as high-priority interrupts? Explain.

9-9 Measuring HiPriISR latency. With the QwikFlash board, jumper the output from **PORTB**, bit 1 (RB1) to the high-priority interrupt input, RB0/INT0.

(a) Write a high-priority interrupt service routine beginning directly at address 0x0008 with the instruction

```
bcf  PORTB,RB1
```

that simply clears the **INT0IF** flag and returns from the interrupt.

(b) Write a mainline program that initializes high-priority interrupts from rising edges on the INT0 input. A

```
bsf  INTCON2,INTEDG0
```

instruction will specify that INT0 interrupts are to occur on rising edges. After initialization has been completed, the mainline loop is to consist of

```
bsf  PORTB,RB1
```

followed by a dozen **nop** instructions. This will ensure that a single-word instruction is being executed when the CPU responds to the interrupt on INT0 caused by this rising edge on RB1. Then branch back to the

```
bsf  PORTB,RB1
```

to repeat the operation endlessly.

(c) Run the program and use a scope to monitor the RB1 pulse width. If zero latency is defined as the pulse width that would arise if the following sequence were executed:

```
bsf  PORTB,RB1
bcf  PORTB,RB1
```

then what is the latency measured by the RB1 pulse width due to the mainline/interrupt interactions?

9-10 Critical regions. Assume that a 1 byte variable called **FLAG** has been defined. Specific bits of **FLAG** are used to pass information from any one of several interrupt handlers back to the mainline code to indicate that its interrupt event has occurred and that the mainline code can take action accordingly, and then clear the specific **FLAG** bit. **FLAG** is thus a variable that is accessed and changed by multiple interrupt handlers as well as the mainline code. Why do the accesses and changes in the mainline code of this shared resource *not* constitute a critical region?