

The concept of an interrupt is something that has expanded in scope over the years. The 80x86 family has only added to the confusion surrounding interrupts by introducing the `int` (software interrupt) instruction. Indeed, different manufacturers have used terms like *exceptions*, *faults*, *aborts*, *traps*, and *interrupts* to describe the phenomena this chapter discusses. Unfortunately, there is no clear consensus as to the exact meaning of these terms. Different authors adopt different terms to their own use. While it is tempting to avoid the use of such misused terms altogether, for the purpose of discussion it would be nice to have a set of well defined terms we can use in this chapter. Therefore, we will pick three of the terms above, interrupts, traps, and exceptions, and define them. This chapter attempts to use the most common meanings for these terms, but don't be surprised to find other texts using them in different contexts.

On the 80x86, there are three types of events commonly known as interrupts: *traps*, *exceptions*, and *interrupts* (hardware interrupts). This chapter will describe each of these forms and discuss their support on the 80x86 CPUs and PC compatible machines.

Although the terms trap and exception are often used synonymously, we will use the term *trap* to denote a programmer initiated and expected transfer of control to a special handler routine. In many respects, a trap is nothing more than a specialized subroutine call. Many texts refer to traps as *software interrupts*. The 80x86 `int` instruction is the main vehicle for executing a trap. Note that traps are usually *unconditional*; that is, when you execute an `int` instruction, control *always* transfers to the procedure associated with the trap. Since traps execute via an explicit instruction, it is easy to determine exactly which instructions in a program will invoke a *trap handling* routine.

An exception is an automatically generated trap (coerced rather than requested) that occurs in response to some exceptional condition. Generally, there isn't a specific instruction associated with an exception<sup>1</sup>, instead, an exception occurs in response to some degenerate behavior of normal 80x86 program execution. Examples of conditions that may *raise* (cause) an exception include executing a division instruction with a zero divisor, executing an illegal opcode, and a memory protection fault. Whenever such a condition occurs, the CPU immediately suspends execution of the current instruction and transfers control to an *exception handler* routine. This routine can decide how to handle the exceptional condition; it can attempt to rectify the problem or abort the program and print an appropriate error message. Although you do not generally execute a specific instruction to cause an exception, as with the software interrupts (traps), execution of some instruction is what causes an exception. For example, you only get a division error when executing a division instruction somewhere in a program.

*Hardware interrupts*, the third category that we will refer to simply as *interrupts*, are program control interruption based on an external hardware event (external to the CPU). These interrupts generally have nothing at all to do with the instructions currently executing; instead, some event, such as pressing a key on the keyboard or a time out on a timer chip, informs the CPU that a device needs some attention. The CPU interrupts the currently executing program, services the device, and then returns control back to the program.

An *interrupt service routine* is a procedure written specifically to handle a trap, exception, or interrupt. Although different phenomenon cause traps, exceptions, and interrupts, the structure of an interrupt service routine, or *ISR*, is approximately the same for each of these.

---

1. Although we will classify the `int` instruction in this category. This is an exception to this rule.

## 17.1 80x86 Interrupt Structure and Interrupt Service Routines (ISRs)

Despite the different causes of traps, exceptions, and interrupts, they share a common format for their handling routines. Of course, these interrupt service routines will perform different activities depending on the source of the invocation, but it is quite possible to write a single interrupt handling routine that processes traps, exceptions, and hardware interrupts. This is rarely done, but the structure of the 80x86 interrupt system allows this. This section will describe the 80x86's interrupt structure and how to write basic interrupt service routines for the 80x86 real mode interrupts.

The 80x86 chips allow up to 256 *vectored* interrupts. This means that you can have up to 256 different sources for an interrupt and the 80x86 will directly call the service routine for that interrupt without any software processing. This is in contrast to *nonvectored* interrupts that transfer control directly to a single interrupt service routine, regardless of the interrupt source.

The 80x86 provides a 256 entry *interrupt vector table* beginning at address 0:0 in memory. This is a 1K table containing 256 4-byte entries. Each entry in this table contains a segmented address that points at the interrupt service routine in memory. Generally, we will refer to interrupts by their index into this table, so interrupt zero's address (vector) is at memory location 0:0, interrupt one's vector is at address 0:4, interrupt two's vector is at address 0:8, etc.

When an interrupt occurs, regardless of source, the 80x86 does the following:

- 1) The CPU pushes the flags register onto the stack.
- 2) The CPU pushes a far return address (segment:offset) onto the stack, segment value first.
- 3) The CPU determines the cause of the interrupt (i.e., the interrupt number) and fetches the four byte interrupt vector from address 0:vector\*4.
- 4) The CPU transfers control to the routine specified by the interrupt vector table entry.

After the completion of these steps, the interrupt service routine takes control. When the interrupt service routine wants to return control, it must execute an `iret` (interrupt return) instruction. The interrupt return pops the far return address and the flags off the stack. Note that executing a far return is insufficient since that would leave the flags on the stack.

There is one minor difference between how the 80x86 processes hardware interrupts and other types of interrupts – upon entry into the hardware interrupt service routine, the 80x86 disables further hardware interrupts by clearing the interrupt flag. Traps and exceptions do not do this. If you want to disallow further hardware interrupts within a trap or exception handler, you must explicitly clear the interrupt flag with a `cli` instruction. Conversely, if you want to allow interrupts within a hardware interrupt service routine, you must explicitly turn them back on with an `sti` instruction. Note that the 80x86's interrupt disable flag only affects hardware interrupts. Clearing the interrupt flag will not prevent the execution of a trap or exception.

ISRs are written like almost any other assembly language procedure except that they return with an `iret` instruction rather than `ret`. Although the distance of the ISR procedure (near vs. far) is usually of no significance, you should make all ISRs *far* procedures. This will make programming easier if you decide to call an ISR directly rather than using the normal interrupt handling mechanism.

Exceptions and hardware interrupts ISRs have a very special restriction: they must *preserve the state of the CPU*. In particular, these ISRs must preserve all registers they modify. Consider the following extremely simple ISR:

```
SimpleISR    proc    far
             mov    ax, 0
             iret
SimpleISR    endp
```

This ISR obviously does *not* preserve the machine state; it explicitly disturbs the value in `ax` and then returns from the interrupt. Suppose you were executing the following code segment when a hardware interrupt transferred control to the above ISR:

```

mov     ax, 5
add     ax, 2

; Suppose the interrupt occurs here.

      puti
      :
      :
```

The interrupt service routine would set the `ax` register to zero and your program would print zero rather than the value five. Worse yet, hardware interrupts are generally *asynchronous*, meaning they can occur at any time and rarely do they occur at the same spot in a program. Therefore, the code sequence above would print seven most of the time; once in a great while it might print zero or two (it will print two if the interrupt occurs between the `mov ax, 5` and `add ax, 2` instructions). Bugs in hardware interrupt service routines are very difficult to find, because such bugs often affect the execution of unrelated code.

The solution to this problem, of course, is to make sure you preserve all registers you use in the interrupt service routine for hardware interrupts and exceptions. Since trap calls are explicit, the rules for preserving the state of the machine in such programs is identical to that for procedures.

Writing an ISR is only the first step to implementing an interrupt handler. You must also initialize the interrupt vector table entry with the address of your ISR. There are two common ways to accomplish this – store the address directly in the interrupt vector table or call DOS and let DOS do the job for you.

Storing the address yourself is an easy task. All you need to do is load a segment register with zero (since the interrupt vector table is in segment zero) and store the four byte address at the appropriate offset within that segment. The following code sequence initializes the entry for interrupt 255 with the address of the `SimpleISR` routine presented earlier:

```

mov     ax, 0
mov     es, ax
pushf
cli
mov     word ptr es:[0ffh*4], offset SimpleISR
mov     word ptr es:[0ffh*4 + 2], seg SimpleISR
popf
```

Note how this code turns off the interrupts while changing the interrupt vector table. This is important if you are patching a hardware interrupt vector because it wouldn't do for the interrupt to occur between the last two `mov` instructions above; at that point the interrupt vector is in an inconsistent state and invoking the interrupt at that point would transfer control to the offset of `SimpleISR` and the segment of the previous interrupt `0FFh` handler. This, of course, would be a disaster. The instructions that turn off the interrupts while patching the vector are unnecessary if you are patching in the address of a trap or exception handler<sup>2</sup>.

Perhaps a better way to initialize an interrupt vector is to use DOS' *Set Interrupt Vector* call. Calling DOS (see "MS-DOS, PC-BIOS, and File I/O" on page 699) with `ah` equal to `25h` provides this function. This call expects an interrupt number in the `al` register and the address of the interrupt service routine in `ds:dx`. The call to MS-DOS that would accomplish the same thing as the code above is

---

2. Strictly speaking, this code sequence does not require the `pushf`, `cli`, and `popf` instructions because interrupt 255 does not correspond to any hardware interrupt on a typical PC machine. However, it is important to provide this example so you're aware of the problem.

```

mov     ax, 25ffh                ;AH=25h, AL=0FFh.
mov     dx, seg SimpleISR       ;Load DS:DX with
mov     ds, dx                  ; address of ISR
lea     dx, SimpleISR
int     21h                     ;Call DOS
mov     ax, dseg                ;Restore DS so it
mov     ds, ax                  ; points back at DSEG.

```

Although this code sequence is a little more complex than poking the data directly into the interrupt vector table, it is safer. Many programs monitor changes made to the interrupt vector table through DOS. If you call DOS to change an interrupt vector table entry, those programs will become aware of your changes. If you circumvent DOS, those programs may not find out that you've patched in your own interrupt and could malfunction.

Generally, it is a very bad idea to patch the interrupt vector table and not restore the original entry after your program terminates. Well behaved programs always save the previous value of an interrupt vector table entry and restore this value before termination. The following code sequences demonstrate how to do this. First, by patching the table directly:

```

mov     ax, 0
mov     es, ax

; Save the current entry in the dword variable IntVectSave:

mov     ax, es:[IntNumber*4]
mov     word ptr IntVectSave, ax
mov     ax, es:[IntNumber*4 + 2]
mov     word ptr IntVectSave+2, ax

; Patch the interrupt vector table with the address of our ISR

pushf                                ;Required if this is a hw interrupt.
cli                                  ; " " " " " " " "

mov     word ptr es:[IntNumber*4], offset OurISR
mov     word ptr es:[IntNumber*4+2], seg OurISR

popf                                  ;Required if this is a hw interrupt.

; Okay, do whatever it is that this program is supposed to do:

:
:

; Restore the interrupt vector entries before quitting:

mov     ax, 0
mov     es, ax

pushf                                ;Required if this is a hw interrupt.
cli                                  ; " " " " " " " "

mov     ax, word ptr IntVectSave
mov     es:[IntNumber*4], ax
mov     ax, word ptr IntVectSave+2
mov     es:[IntNumber*4 + 2], ax

popf                                  ;Required if this is a hw interrupt.
:
:

```

If you would prefer to call DOS to save and restore the interrupt vector table entries, you can obtain the address of an existing interrupt table entry using the DOS *Get Interrupt Vector* call. This call, with *ah*=35h, expects the interrupt number in *al*; it returns the existing vector for that interrupt in the *es:bx* registers. Sample code that preserves the interrupt vector using DOS is

```

; Save the current entry in the dword variable IntVectSave:

        mov     ax, 3500h + IntNumber           ;AH=35h, AL=Int #.
        int     21h
        mov     word ptr IntVectSave, bx
        mov     word ptr IntVectSave+2, es

; Patch the interrupt vector table with the address of our ISR

        mov     dx, seg OurISR
        mov     ds, dx
        lea    dx, OurISR
        mov     ax, 2500h + IntNumber           ;AH=25, AL=Int #.
        int     21h

; Okay, do whatever it is that this program is supposed to do:
        .
        .
        .

; Restore the interrupt vector entries before quitting:

        lds    bx, IntVectSave
        mov     ax, 2500h+IntNumber             ;AH=25, AL=Int #.
        int     21h
        .
        .
        .

```

---

## 17.2 Traps

A trap is a software-invoked interrupt. To execute a trap, you use the 80x86 `int` (software interrupt) instruction<sup>3</sup>. There are only two primary differences between a trap and an arbitrary far procedure call: the instruction you use to call the routine (`int` vs. `call`) and the fact that a trap pushes the flags on the stack so you must use the `iret` instruction to return from it. Otherwise, there really is no difference between a trap handler's code and the body of a typical far procedure.

The main purpose of a trap is to provide a fixed subroutine that various programs can call without having to actually know the run-time address. MS-DOS is the perfect example. The `int 21h` instruction is an example of a trap invocation. Your programs do not have to know the actual memory address of DOS' entry point to call DOS. Instead, DOS patches the interrupt 21h vector when it loads into memory. When you execute `int 21h`, the 80x86 automatically transfers control to DOS' entry point, wherever in memory that happens to be.

There is a long lists of support routines that use the trap mechanism to link application programs to themselves. DOS, BIOS, the mouse drivers, and Netware™ are a few examples. Generally, you would use a trap to call a *resident program* function. Resident programs (see "Resident Programs" on page 1025) load themselves into memory and remain resident once they terminate. By patching an interrupt vector to point at a subroutine within the resident code, other programs that run after the resident program terminates can call the resident subroutines by executing the appropriate `int` instruction.

Most resident programs do *not* use a separate interrupt vector entry for each function they provide. Instead, they usually patch a *single* interrupt vector and transfer control to an appropriate routine using a *function number* that the caller passes in a register. By convention, most resident programs expect the function number in the `ah` register. A typical trap handler would execute a case statement on the value in the `ah` register and transfer control to the appropriate handler function.

---

3. You can also simulate an `int` instruction by pushing the flags and executing a far call to the trap handler. We will consider this mechanism later on.

Since trap handlers are virtually identical to far procedures in terms of use, we will not discuss traps in any more detail here. However, the text chapter will explore this subject in greater depth when it discusses resident programs.

---

## 17.3 Exceptions

Exceptions occur (are *raised*) when an abnormal condition occurs during execution. There are fewer than eight possible exceptions on machines running in real mode. Protected mode execution provides many others, but we will not consider those here, we will only consider those exceptions interesting to those working in real mode<sup>4</sup>.

Although exception handlers are user defined, the 80x86 hardware defines the exceptions that can occur. The 80x86 also assigns a fixed interrupt number to each of the exceptions. The following sections describe each of these exceptions in detail.

In general, an exception handler should preserve all registers. However, there are several special cases where you may want to tweak a register value before returning. For example, if you get a bounds violation, you may want to modify the value in the register specified by the bound instruction before returning. Nevertheless, you should not arbitrarily modify registers in an exception handling routine unless you intend to immediately abort the execution of your program.

---

### 17.3.1 Divide Error Exception (INT 0)

This exception occurs whenever you attempt to divide a value by zero or the quotient does not fit in the destination register when using the `div` or `idiv` instructions. Note that the FPU's `fdiv` and `fdivr` instructions do *not* raise this exception.

MS-DOS provides a generic divide exception handler that prints a message like "divide error" and returns control to MS-DOS. If you want to handle division errors yourself, you must write your own exception handler and patch the address of this routine into location 0:0.

On 8086, 8088, 80186, and 80188 processors, the return address on the stack points at the next instruction after the divide instruction. On the 80286 and later processors, the return address points at the beginning of the divide instruction (include any prefix bytes that appear). When a divide exception occurs, the 80x86 registers are unmodified; that is, they contain the values they held when the 80x86 first executed the `div` or `idiv` instruction.

When a divide exception occurs, there are three reasonable things you can attempt: abort the program (the easy way out), jump to a section of code that attempts to continue program execution in view of the error (e.g., as the user to reenter a value), or attempt to figure out why the error occurred, correct it, and reexecute the division instruction. Few people choose this last alternative because it is so difficult.

---

### 17.3.2 Single Step (Trace) Exception (INT 1)

The single step exception occurs after every instruction if the `trace` bit in the flags register is equal to one. Debuggers and other programs will often set this flag so they can trace the execution of a program.

When this exception occurs, the return address on the stack is the address of the *next* instruction to execute. The trap handler can decode this opcode and decide how to proceed. Most debuggers use the trace exception to check for *watchpoints* and other events that change dynamically during program execution. Debuggers that use the trace excep-

---

4. For more details on exceptions in protected mode, see the bibliography.

tion for single stepping often *disassemble* the next instruction using the return address on the stack as a pointer to that instruction's opcode bytes.

Generally, a single step exception handler should preserve *all* 80x86 registers and other state information. However, you will see an interesting use of the trace exception later in this text where we will purposely modify register values to make one instruction behave like another (see “The PC Keyboard” on page 1153).

Interrupt one is also shared by the debugging exceptions capabilities of 80386 and later processors. These processors provide on-chip support via *debugging registers*. If some condition occurs that matches a value in one of the debugging registers, the 80386 and later CPUs will generate a debugging exception that uses interrupt vector one.

### 17.3.3 Breakpoint Exception (INT 3)

The breakpoint exception is actually a trap, not an exception. It occurs when the CPU executes an int 3 instruction. However, we will consider it an exception since programmers rarely put int 3 instructions directly into their programs. Instead, a debugger like Codeview often manages the placement and removal of int 3 instructions.

When the 80x86 calls a breakpoint exception handling routine, the return address on the stack is the address of the next instruction after the breakpoint opcode. Note, however, that there are actually *two* int instructions that transfer control through this vector. Generally, though, it is the one-byte int 3 instruction whose opcode is 0cch; otherwise it is the two byte equivalent: 0cdh, 03h.

### 17.3.4 Overflow Exception (INT 4/INTO)

The overflow exception, like int 3, is technically a trap. The CPU only raises this exception when you execute an into instruction and the overflow flag is set. If the overflow flag is clear, the into instruction is effectively a nop, if the overflow flag is set, into behaves like an int 4 instruction. Programmers can insert an into instruction after an integer computation to check for an arithmetic overflow. Using into is equivalent to the following code sequence:

```

    « Some integer arithmetic code »
        jno     GoodCode
        int    4
GoodCode:

```

One big advantage to the into instruction is that it does not flush the pipeline or prefetch queue if the overflow flag is not set. Therefore, using the into instruction is a good technique if you provide a single overflow handler (that is, you don't have some special code for each sequence where an overflow could occur).

The return address on the stack is the address of the next instruction after into. Generally, an overflow handler does not return to that address. Instead, it will usually abort the program or pop the return address and flags off the stack and attempt the computation in a different way.

### 17.3.5 Bounds Exception (INT 5/BOUND)

Like into, the bound instruction (see “The INT, INTO, BOUND, and IRET Instructions” on page 292) will cause a conditional exception. If the specified register is outside the specified bounds, the bound instruction is equivalent to an int 5 instruction; if the register is within the specified bounds, the bound instruction is effectively a nop.

The return address that bound pushes is the address of the bound instruction itself, not the instruction following bound. If you return from the exception without modifying the

value in the register (or adjusting the bounds), you will generate an infinite loop because the code will reexecute the bound instruction and repeat this process over and over again.

One sneaky trick with the bound instruction is to generate a global minimum and maximum for an array of signed integers. The following code demonstrates how you can do this:

```

; This program demonstrates how to compute the minimum and maximum values
; for an array of signed integers using the bound instruction

        .xlist
        .286
        include    stdlib.a
        includelib stdlib.lib
        .list

dseg          segment    para public 'data'

; The following two values contain the bounds for the BOUND instruction.

LowerBound    word      ?
UpperBound    word      ?

; Save the INT 5 address here:

OldInt5       dword     ?

; Here is the array we want to compute the minimum and maximum for:

Array         word      1, 2, -5, 345, -26, 23, 200, 35, -100, 20, 45
              word      62, -30, -1, 21, 85, 400, -265, 3, 74, 24, -2
              word      1024, -7, 1000, 100, -1000, 29, 78, -87, 60
ArraySize     =          ($-Array)/2

dseg          ends

cseg          segment    para public 'code'
              assume     cs:cseg, ds:dseg

; Our interrupt 5 ISR. It compares the value in AX with the upper and
; lower bounds and stores AX in one of them (we know AX is out of range
; by virtue of the fact that we are in this ISR).
;
; Note: in this particular case, we know that DS points at dseg, so this
; ISR will get cheap and not bother reloading it.
;
; Warning: This code does not handle the conflict between bound/int5 and
; the print screen key. Pressing prtsc while executing this code may
; produce incorrect results (see the text).

BoundISR      proc        near
              cmp         ax, LowerBound
              jl          NewLower

; Must be an upper bound violation.

              mov         UpperBound, ax
              iret

NewLower:     mov         LowerBound, ax
              iret
BoundISR      endp

Main          proc
              mov         ax, dseg
              mov         ds, ax
              meminit

```



```

; Begin by patching in the address of our ISR into int 5's vector.

        mov     ax, 0
        mov     es, ax
        mov     ax, es:[5*4]
        mov     word ptr OldInt5, ax
        mov     ax, es:[5*4 + 2]
        mov     word ptr OldInt5+2, ax

        mov     word ptr es:[5*4], offset BoundISR
        mov     es:[5*4 + 2], cs

; Okay, process the array elements. Begin by initializing the upper
; and lower bounds values with the first element of the array.

        mov     ax, Array
        mov     LowerBound, ax
        mov     UpperBound, ax

; Now process each element of the array:

        mov     bx, 2                ;Start with second element.
        mov     cx, ArraySize
GetMinMax:
        mov     ax, Array[bx]
        bound   ax, LowerBound
        add     bx, 2                ;Move on to next element.
        loop   GetMinMax            ;Repeat for each element.

        printf
        byte   "The minimum value is %d\n"
        byte   "The maximum value is %d\n",0
        dword  LowerBound, UpperBound

; Okay, restore the interrupt vector:

        mov     ax, 0
        mov     es, ax
        mov     ax, word ptr OldInt5
        mov     es:[5*4], ax
        mov     ax, word ptr OldInt5+2
        mov     es:[5*4+2], ax

Quit:   ExitPgm                    ;DOS macro to quit program.
Main    endp

cseg    ends

sseg    segment para stack 'stack'
stk     db    1024 dup ("stack ")
sseg    ends

zzzzzzseg    segment para public 'zzzzzz'
LastBytes   db    16 dup (?)
zzzzzzseg    ends
end        Main

```

If the array is large and the values appearing in the array are relatively random, this code demonstrates a fast way to determine the minimum and maximum values in the array. The alternative, comparing each element against the upper and lower bounds and storing the value if outside the range, is generally a slower approach. True, if the bound instruction causes a trap, this is *much* slower than the compare and store method. However, in a large array with random values, the bounds violation will rarely occur. Most of the time the bound instruction will execute in 7-13 clock cycles and it will not flush the pipeline or the prefetch queue<sup>5</sup>.

**Warning:** IBM, in their infinite wisdom, decided to use int 5 as the *print screen* operation. The default int 5 handler will dump the current contents of the screen to the printer. This has two implications for those who would like to use the bound instruction in their programs. First, if you do not install your own int 5 handler and you execute a bound instruction that generates a bound exception, you will cause the machine to print the contents of the screen. Second, if you press the PrtSc key with your int 5 handler installed, BIOS will invoke your handler. The former case is a programming error, but this latter case means you have to make your bounds exception handler a little smarter. It should look at the byte pointed at by the return address. If this is an int 5 instruction opcode (0cdh), then you need to call the original int 5 handler, or simply return from interrupt (do you want them pressing the PrtSc key at that point?). If it is not an int 5 opcode, then this exception was probably raised by the bound instruction. Note that when executing a bound instruction the return address may not be pointing directly at a bound opcode (0c2h). It may be pointing at a prefix byte to the bound instruction (e.g., segment, addressing mode, or size override). Therefore, it is best to check for the int 5 opcode.

### 17.3.6 Invalid Opcode Exception (INT 6)

The 80286 and later processors raise this exception if you attempt to execute an opcode that does not correspond to a legal 80x86 instruction. These processors also raise this exception if you attempt to execute a bound, lds, les, lidt, or other instruction that requires a memory operand but you specify a register operand in the mod/reg/rm field of the mod/reg/rm byte.

The return address on the stack points at the illegal opcode. By examining this opcode, you can extend the instruction set of the 80x86. For example, you could run 80486 code on an 80386 processor by providing subroutines that mimic the extra 80486 instructions (like bswap, cmpxchg, etc.).

### 17.3.7 Coprocessor Not Available (INT 7)

The 80286 and later processors raise this exception if you attempt to execute an FPU (or other coprocessor) instruction without having the coprocessor installed. You can use this exception to simulate the coprocessor in software.

On entry to the exception handler, the return address points at the coprocessor opcode that generated the exception.

## 17.4 Hardware Interrupts

Hardware interrupts are the form most engineers (as opposed to PC programmers) associate with the term *interrupt*. We will adopt this same strategy henceforth and will use the non-modified term “interrupt” to mean a hardware interrupt.

On the PC, interrupts come from many different sources. The primary sources of interrupts, however, are the PC's timer chip, keyboard, serial ports, parallel ports, disk drives, CMOS real-time clock, mouse, sound cards, and other peripheral devices. These devices connect to an Intel 8259A programmable interrupt controller (PIC) that prioritizes the interrupts and interfaces with the 80x86 CPU. The 8259A chip adds considerable complexity to the software that processes interrupts, so it makes perfect sense to discuss the PIC first, before trying to describe how the interrupt service routines have to deal with it. Afterwards, this section will briefly describe each device and the conditions under which

5. Note that on the 80486 and later processors, the bound instruction may actually be slower than the corresponding straight line code.

it interrupts the CPU. This text will fully describe many of these devices in later chapters, so this chapter will not go into a lot of detail except when discussing the timer interrupt.

### 17.4.1 The 8259A Programmable Interrupt Controller (PIC)

The 8259A (8259<sup>6</sup> or PIC, hereafter) programmable interrupt controller chip accepts interrupts from up to eight different devices. If any one of the devices requests service, the 8259 will toggle an interrupt output line (connected to the CPU) and pass a programmable interrupt vector to the CPU. You can *cascade* the device to support up to 64 devices by connecting nine 8259s together: eight of the devices with eight inputs each whose outputs become the eight inputs of the ninth device. A typical PC uses two of these devices to provide 15 interrupt inputs (seven on the *master* PIC with the eight input coming from the *slave* PIC to process its eight inputs)<sup>7</sup>. The sections following this one will describe the devices connected to each of those inputs, for now we will concentrate on what the 8259 does with those inputs. Nevertheless, for the sake of discussion, the following table lists the interrupt sources on the PC:

**Table 66: 8259 Programmable Interrupt Controller Inputs**

Input on 8259	80x86 INT	Device
IRQ 0	8	Timer chip
IRQ 1	9	Keyboard
IRQ 2	0Ah	Cascade for controller 2 (IRQ 8-15)
IRQ 3	0Bh	Serial port 2
IRQ 4	0Ch	Serial port 1
IRQ 5	0Dh	Parallel port 2 in AT, reserved in PS/2 systems
IRQ 6	0Eh	Diskette drive
IRQ 7	0Fh	Parallel port 1
IRQ 8/0	70h	Real-time clock
IRQ 9/1	71h	CGA vertical retrace (and other IRQ 2 devices)
IRQ 10/2	72h	Reserved
IRQ 11/3	73h	Reserved
IRQ 12/4	74h	Reserved in AT, auxiliary device on PS/2 systems
IRQ 13/5	75h	FPU interrupt
IRQ 14/6	76h	Hard disk controller
IRQ 15/7	77h	Reserved

The 8259 PIC is a very complex chip to program. Fortunately, all of the hard stuff has already been done for you by the BIOS when the system boots. We will not discuss how to initialize the 8259 in this text because that information is only useful to those writing operating systems like Linux, Windows, or OS/2. If you want your interrupt service routines to run correctly under DOS or any other OS, you must not reinitialize the PIC.

The PICs interface to the system through four I/O locations: ports 20h/0A0h and 21h/0A1h. The first address in each pair is the address of the master PIC (IRQ 0-7), the

6. The original 8259 was designed for Intel's 8080 system. The 8259A provided support for the 80x86 and some other features. Since almost no one uses 8259 chips anymore, this text will use the generic term 8259.

7. The original IBM PC and PC/XT machines only supported eight interrupts via one 8259 chip. IBM, and virtually all clone manufacturers, added the second PIC in PC/AT and later designs.

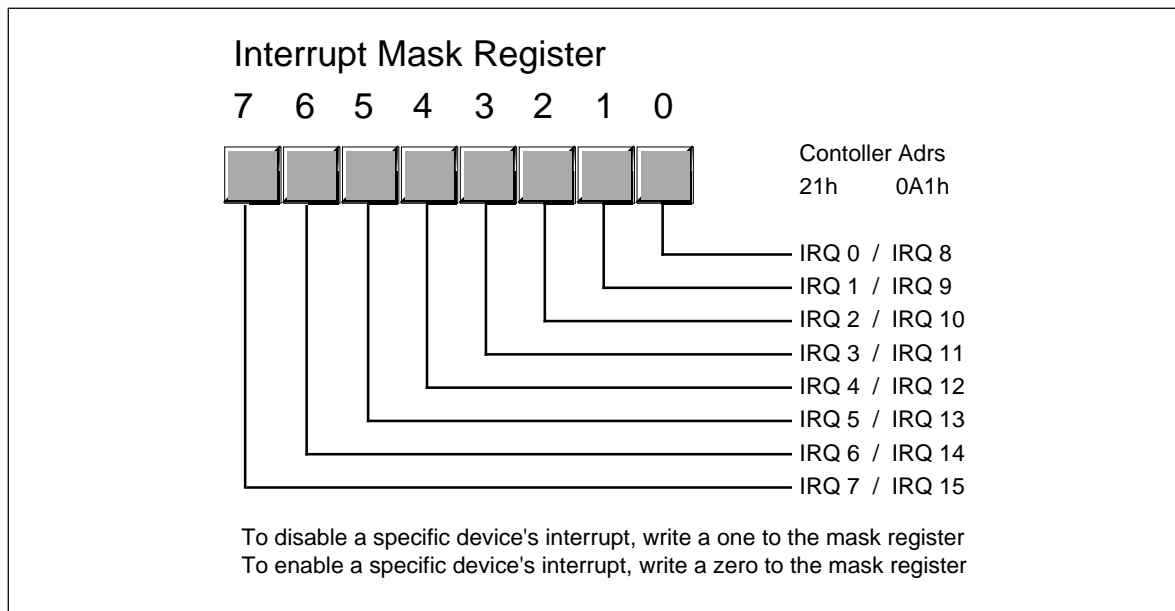


Figure 17.1 8259 Interrupt Mask Register

second address in each pair corresponds to the slave PIC (IRQ 8-15). Port 20h/0A0h is a read/write location to which you write PIC commands and read PIC status, we will refer to this as the *command register* or the *status register*. The command register is write only, the status register is read only. They just happen to share the same I/O location. The read/write lines on the PIC determine which register the CPU accesses. Port 21h/0A1h is a read/write location that contains the interrupt mask register, we will refer to this as the *mask register*. Choose the appropriate address depending upon which interrupt controller you want to use.

The interrupt mask register is an eight bit register that lets you individually enable and disable interrupts from devices on the system. This is similar to the actions of the `cli` and `sti` instructions, but on a device by device basis. Writing a zero to the corresponding bit *enables* that device's interrupts. Writing a one *disables* interrupts from the affected device. Note that this is non-intuitive. Figure 17.1 provides the layout of the interrupt mask register.

When changing bits in the mask register, it is important that you not simply load `al` with a value and output it directly to the mask register port. Instead, you should read the mask register and then logically or in or and out the bits you want to change; finally, you can write the output back to the mask register. The following code sequence enables COM1: interrupts without affecting any others:

```

in      al, 21h           ;Read existing bits.
and     al, 0efh         ;Turn on IRQ 4 (COM1).
out     21h, al          ;Write result back to PIC.

```

The command register provides lots of options, but there are only three commands you would want to execute on this chip that are compatible with the BIOS' initialization of the 8259: sending an end of interrupt command and sending one of two read status register commands.

One a specific interrupt occurs, the 8259 masks all further interrupts from that device until it receives an *end of interrupt* signal from the interrupt service routine. On PCs running DOS, you accomplish this by writing the value 20h to the command register. The following code does this:

```

mov     al, 20h
out     20h, al           ;Port 0A0h if IRQ 8-15.

```

You must send exactly one end of interrupt command to the PIC for each interrupt you service. If you do not send the end of interrupt command, the PIC will not honor any more interrupts from that device; if you send two or more end of interrupt commands, there is the possibility that you will accidentally acknowledge a new interrupt that may be pending and you will lose that interrupt.

For some interrupt service routines you write, your ISR will not be the only ISR that an interrupt invokes. For example, the PC's BIOS provides an ISR for the timer interrupt that maintains the time of day. If you patch into the timer interrupt, you will need to call the PC BIOS' timer ISR so the system can properly maintain the time of day and handle other timing related chores (see "Chaining Interrupt Service Routines" on page 1010). However, the BIOS' timer ISR outputs the end of interrupt command. Therefore, you should not output the end of interrupt command yourself, otherwise the BIOS will output a second end of interrupt command and you may lose an interrupt in the process.

The other two commands you can send the 8259 let you select whether to read the *in-service register* (ISR) or the *interrupt request register* (IRR). The in-service register contains set bits for each active ISR (because the 8259 allows prioritized interrupts, it is quite possible that one ISR has been interrupted by a higher priority ISR). The interrupt request register contains set bits in corresponding positions for interrupts that have not yet been serviced (probably because they are a lower priority interrupt than the interrupt currently being serviced by the system). To read the in-service register, you would execute the following statements:

```
; Read the in-service register in PIC #1 (at I/O address 20h)

        mov     al, 0bh
        out     20h, al
        in      al, 20h
```

To read the interrupt request register, you would use the following code:

```
; Read the interrupt request register in PIC #1 (at I/O address 20h)

        mov     al, 0ah
        out     20h, al
        in      al, 20h
```

Writing any other values to the command port may cause your system to malfunction.

---

## 17.4.2 The Timer Interrupt (INT 8)

The PC's motherboard contains an 8254 compatible timer chip. This chip contains three timer channels, one of which generates interrupts every 55 msec (approximately). This is about once every  $1/18.2$  seconds. You will often hear this interrupt referred to as the "eighteenth second clock." We will simply call it the timer interrupt.

The timer interrupt vector is probably the most commonly patched interrupt in the system. It turns out there are *two* timer interrupt vectors in the system. Int 8 is the hardware vector associated with the timer interrupt (since it comes in on IRQ 0 on the PIC). Generally, you should *not* patch this interrupt if you want to write a timer ISR. Instead, you should patch the second timer interrupt, interrupt 1ch. The BIOS' timer interrupt handler (int 8) executes an int 1ch instruction before it returns. This gives a user patched routine access to the timer interrupt. Unless you are willing to duplicate the BIOS and DOS timer code, you should never completely replace the existing timer ISR with one of your own, you should always ensure that the BIOS and DOS ISRs execute in addition to your ISR. Patching into the int 1ch vector is the easiest way to do this.

Even replacing the int 1ch vector with a pointer to your ISR is very dangerous. The timer interrupt service routine is the one most commonly patched by various resident programs (see "Resident Programs" on page 1025). By simply writing the address of your ISR into the timer interrupt vector, you may disable such resident programs and cause your

system to malfunction. To solve this problem, you need to create an *interrupt chain*. For more details, see the section “Chaining Interrupt Service Routines” on page 1010.

By default the timer interrupt is always enabled on the interrupt controller chip. Indeed, disabling this interrupt may cause your system to crash or otherwise malfunction. At the very least, your system will not maintain the correct time if you disable the timer interrupt.

---

### 17.4.3 The Keyboard Interrupt (INT 9)

The keyboard microcontroller on the PC’s motherboard generates *two* interrupts on each keystroke – one when you press a key and one when you release it. This is on IRQ 1 on the master PIC. The BIOS responds to this interrupt by reading the keyboard’s *scan code*, converting this to an ASCII character, and storing the scan and ASCII codes away in the system *type ahead buffer*.

By default, this interrupt is always enabled. If you disable this interrupt, the system will not be able to respond to any keystrokes, including ctrl-alt-del. Therefore, your programs should always reenable this interrupt if they ever disable it.

For more information on the keyboard interrupt, see “The PC Keyboard” on page 1153.

---

### 17.4.4 The Serial Port Interrupts (INT 0Bh and INT 0Ch)

The PC uses two interrupts, IRQ 3 and IRQ 4, to support interrupt driven serial communications. The 8250 (or compatible) serial communications controller chip (SCC) generates an interrupt in one of four situations: a character arriving over the serial line, the SCC finishes the transmission of a character and is requesting another, an error occurs, or a status change occurs. The SCC activates the same interrupt line (IRQ 3 or 4) for all four interrupt sources. The interrupt service routine is responsible for determining the exact nature of the interrupt by interrogating the SCC.

By default, the system disables IRQ 3 and IRQ 4. If you install a serial ISR, you will need to clear the interrupt mask bit in the 8259 PIC before it will respond to interrupts from the SCC. Furthermore, the SCC design includes its own interrupt mask. You will need to enable the interrupt masks on the SCC chip as well. For more information on the SCC, see “The PC Serial Ports” on page 1223.

---

### 17.4.5 The Parallel Port Interrupts (INT 0Dh and INT 0Fh)

The parallel port interrupts are an enigma. IBM designed the original system to allow two parallel port interrupts and then promptly designed a printer interface card that didn’t support the use of interrupts. As a result, almost no DOS based software today uses the parallel port interrupts (IRQ 5 and IRQ 7). Indeed, on the PS/2 systems IBM reserved IRQ5 which they formerly used for LPT2:.

However, these interrupts have not gone to waste. Many devices which IBM’s engineers couldn’t even conceive when designing the first PC have made good use of these interrupts. Examples include SCSI cards and sound cards. Many devices today include “interrupt jumpers” that let you select IRQ 5 or IRQ 7 when installing the device.

Since IRQ 5 and IRQ 7 find such little use as parallel port interrupts, we will effectively ignore the “parallel port interrupts” in this text.

---

## 17.4.6 The Diskette and Hard Drive Interrupts (INT 0Eh and INT 76h)

The floppy and hard disk drives generate interrupts at the completion of a disk operation. This is a very useful feature for multitasking systems like OS/2, Linux, or Windows. While the disk is reading or writing data, the CPU can go execute instructions for another process. When the disk finishes the read or write operation, it interrupts the CPU so it can resume the original task.

While managing the disk drives would be an interesting topic to cover in this text, this book is already long enough. Therefore, this text will avoid discussing the disk drive interrupts (IRQ 6 and IRQ 14) in the interest of saving some space. There are many texts that cover low level disk I/O in assembly language, see the bibliography for details.

By default, the floppy and hard disk interrupts are always enabled. You should not change this status if you intend to use the disk drives on your system.

---

## 17.4.7 The Real-Time Clock Interrupt (INT 70h)

PC/AT and later machines included a CMOS real-time clock. This device is capable of generating timer interrupts in multiples of 976  $\mu$ sec (let's call it 1 msec). By default, the real-time clock interrupt is disabled. You should only enable this interrupt if you have an int 70h ISR installed.

---

## 17.4.8 The FPU Interrupt (INT 75h)

The 80x87 FPU generates an interrupt whenever a floating point exception occurs. On CPUs with built-in FPUs (80486DX and better) there is a bit in one of the control register you can set to simulate a vectored interrupt. BIOS generally initializes such bits for compatibility with existing systems.

By default, BIOS disables the FPU interrupt. Most programs that use the FPU explicitly test the FPU's status register to determine if an error occurs. If you want to allow FPU interrupts, you must enable the interrupts on the 8259 *and* on the 80x87 FPU.

---

## 17.4.9 Nonmaskable Interrupts (INT 2)

The 80x86 chips actually provide *two* interrupt input pins. The first is the *maskable* interrupt. This is the pin to which the 8259 PIC connects. This interrupt is maskable because you can enable or disable it with the *cli* and *sti* instructions. The *nonmaskable* interrupt, as its name implies, cannot be disabled under software control. Generally, PCs use this interrupt to signal a memory parity error, although certain systems use this interrupt for other purposes as well. Many older PC systems connect the FPU to this interrupt.

This interrupt cannot be masked, so it is always enabled by default.

---

## 17.4.10 Other Interrupts

As mentioned in the section on the 8259 PIC, there are several interrupts reserved by IBM. Many systems use the reserved interrupts for the mouse or for other purposes. Since such interrupts are inherently system dependent, we will not describe them here.

## 17.5 Chaining Interrupt Service Routines

Interrupt service routines come in two basic varieties – those that need exclusive access to an interrupt vector and those that must share an interrupt vector with several other ISRs. Those in the first category include error handling ISRs (e.g., divide error or overflow) and certain device drivers. The serial port is a good example of a device that rarely has more than one ISR associated with it at any one given time<sup>8</sup>. The timer, real-time clock, and keyboard ISRs generally fall into the latter category. It is not at all uncommon to find several ISRs in memory sharing each of these interrupts.

Sharing an interrupt vector is rather easy. All an ISR needs to do to share an interrupt vector is to save the old interrupt vector when installing the ISR (something you need to do anyway, so you can restore the interrupt vector when your code terminates) and then call the original ISR before or after you do your own ISR processing. If you've saved away the address of the original ISR in the dseg double word variable `OldIntVect`, you can call the original ISR with the following code:

```
; Presumably, DS points at DSEG at this point.

        pushf                ;Simulate an INT instruction by pushing
        call   OldIntVect    ; the flags and making a far call.
```

Since `OldIntVect` is a dword variable, this code generates a far call to the routine whose segmented address appears in the `OldIntVect` variable. This code does *not* jump to the location of the `OldIntVect` variable.

Many interrupt service routines do not modify the `ds` register to point at a local data segment. In fact, some simple ISRs do not change any of the segment registers. In such cases it is common to put any necessary variables (especially the old segment value) directly in the code segment. If you do this, your code could *jump* directly to the original ISR rather than calling it. To do so, you would just use the code:

```
MyISR      proc      near
            .
            .
            jmp      cs:OldIntVect
MyISR      endp

OldIntVect dword    ?
```

This code sequence passes along your ISR's flags and return address as the flag and return address values to the original ISR. This is fine, when the original ISR executes the `iret` instruction, it will return directly to the interrupted code (assuming it doesn't pass control to some other ISR in the chain).

The `OldIntVect` variable *must* be in the code segment if you use this technique to transfer control to the original ISR. After all, when you executing the `jmp` instruction above, you must have already restored the state of the CPU, including the `ds` register. Therefore, you have no idea what segment `ds` is pointing at, and it probably isn't pointing at your local data segment. Indeed, the only segment register whose value is known to you is `cs`, so you must keep the vector address in your code segment.

The following simple program demonstrates interrupt chaining. This short program patches into the `int 1ch` vector. The ISR counts off seconds and notifies the main program as each second passes. The main program prints a short message every second. When 10 seconds have expired, this program removes the ISR from the interrupt chain and terminates.

```
; TIMER.ASM
; This program demonstrates how to patch into the int 1Ch timer interrupt
; vector and create an interrupt chain.
```

---

8. There is no reason this has to be this way, it's just that most people rarely run two programs at the same time which must both be accessing the serial port.



```

        .xlist
        .286
        include    stdlib.a
        includelib stdlib.lib
        .list

dseg          segment para public 'data'

; The TIMERISR will update the following two variables.
; It will update the MSEC variable every 55 ms.
; It will update the TIMER variable every second.

MSEC          word    0
TIMER         word    0

dseg          ends

cseg          segment para public 'code'
              assume  cs:cseg, ds:dseg

; The OldInt1C variable must be in the code segment because of the
; way TimerISR transfers control to the next ISR in the int 1Ch chain.

OldInt1C      dword   ?

; The timer interrupt service routine.
; This guy increment MSEC variable by 55 on every interrupt.
; Since this interrupt gets called every 55 msec (approx) the
; MSEC variable contains the current number of milliseconds.
; When this value exceeds 1000 (one second), the ISR subtracts
; 1000 from the MSEC variable and increments TIMER by one.

TimerISR      proc    near
              push   ds
              push   ax
              mov    ax, dseg
              mov    ds, ax

              mov    ax, MSEC
              add    ax, 55          ;Interrupt every 55 msec.
              cmp    ax, 1000
              jnb   SetMSEC
              inc    Timer          ;A second just passed.
              sub    ax, 1000      ;Adjust MSEC value.
SetMSEC:      mov    MSEC, ax
              pop    ax
              pop    ds
              jmp    cseg:OldInt1C ;Transfer to original ISR.
TimerISR      endp

Main          proc
              mov    ax, dseg
              mov    ds, ax
              meminit

; Begin by patching in the address of our ISR into int 1ch's vector.
; Note that we must turn off the interrupts while actually patching
; the interrupt vector and we must ensure that interrupts are turned
; back on afterwards; hence the cli and sti instructions. These are
; required because a timer interrupt could come along between the two
; instructions that write to the int 1Ch interrupt vector. This would
; be a big mess.

              mov    ax, 0
              mov    es, ax
              mov    ax, es:[1ch*4]
              mov    word ptr OldInt1C, ax
              mov    ax, es:[1ch*4 + 2]

```

```

                                mov     word ptr OldInt1C+2, ax

                                cli
                                mov     word ptr es:[1Ch*4], offset TimerISR
                                mov     es:[1Ch*4 + 2], cs
                                sti

; Okay, the ISR updates the TIMER variable every second.
; Continuously print this value until ten seconds have
; elapsed. Then quit.

TimerLoop:
                                mov     Timer, 0
                                printf  "Timer = %d\n",0
                                byte    "Timer = %d\n",0
                                dword   Timer
                                cmp     Timer, 10
                                jbe     TimerLoop

; Okay, restore the interrupt vector. We need the interrupts off
; here for the same reason as above.

                                mov     ax, 0
                                mov     es, ax
                                cli
                                mov     ax, word ptr OldInt1C
                                mov     es:[1Ch*4], ax
                                mov     ax, word ptr OldInt1C+2
                                mov     es:[1Ch*4+2], ax
                                sti

Quit:                            ExitPgm                ;DOS macro to quit program.
Main                             endp

cseg                              ends

sseg                             segment para stack 'stack'
stk                              db     1024 dup ("stack ")
sseg                             ends

zzzzzzseg                       segment para public 'zzzzzz'
LastBytes                       db     16 dup (?)
zzzzzzseg                       ends
end                               Main

```

---

## 17.6 Reentrancy Problems

A minor problem develops with developing ISRs, what happens if you enable interrupts while in an ISR and a second interrupt from the same device comes along? This would interrupt the ISR and then *reenter* the ISR from the beginning. Many applications do not behave properly under these conditions. An application that can properly handle this situation is said to be *reentrant*. Code segments that do not operate properly when reentered are *nonreentrant*.

Consider the TIMER.ASM program in the previous section. This is an example of a nonreentrant program. Suppose that while executing the ISR, it is interrupted at the following point:

```

TimerISR      proc     near
              push    ds
              push    ax
              mov     ax, dseg
              mov     ds, ax

              mov     ax, MSEC
              add     ax, 55          ;Interrupt every 55 msec.
              cmp     ax, 1000
              jb     SetMSEC

```

```

; <<<<< Suppose the interrupt occurs at this point >>>>>

                inc      Timer      ;A second just passed.
                sub      ax, 1000   ;Adjust MSEC value.
SetMSEC:        mov      MSEC, ax
                pop      ax
                pop      ds
                jmp      cseg:OldInt1C ;Transfer to original ISR.
TimerISR       endp

```

Suppose that, on the first invocation of the interrupt, MSEC contains 950 and Timer contains three. If a second interrupt occurs and the specified point above, ax will contain 1005. So the interrupt suspends the ISR and reenters it from the beginning. Note that TimerISR is nice enough to preserve the ax register containing the value 1005. When the second invocation of TimerISR executes, it finds that MSEC still contains 950 because the first invocation has yet to update MSEC. Therefore, it adds 55 to this value, determines that it exceeds 1000, increments Timer (it becomes four) and then stores five into MSEC. Then it returns (by jumping to the next ISR in the int 1ch chain). Eventually, control returns the first invocation of the TimerISR routine. At this time (less than 55 msec after updating Timer by the second invocation) the TimerISR code increments the Timer variable again and updates MSEC to five. The problem with this sequence is that it has incremented the Timer variable twice in less than 55 msec.

Now you might argue that hardware interrupts always clear the interrupt disable flag so it would not be possible for this interrupt to be reentered. Furthermore, you might argue that this routine is so short, it would never take more than 55 msec to get to the noted point in the code above. However, you are forgetting something: some other timer ISR could be in the system that calls *your* code after it is done. That code could take 55 msec and just happen to turn the interrupts back on, making it perfectly possible that your code could be reentered.

The code between the mov ax, MSEC and mov MSEC, ax instructions above is called a *critical region* or *critical section*. A program must not be reentered while it is executing in a critical region. Note that having critical regions does not mean that a program is not reentrant. Most programs, even those that are reentrant, have various critical regions. The key is to prevent an interrupt that could cause a critical region to be reentered while in that critical region. The easiest way to prevent such an occurrence is to *turn off the interrupts* while executing code in a critical section. We can easily modify the TimerISR to do this with the following code:

```

TimerISR       proc      near
                push    ds
                push    ax
                mov     ax, dseg
                mov     ds, ax

; Beginning of critical section, turn off interrupts.

                pushf                    ;Preserve current I flag state.
                cli                      ;Make sure interrupts are off.

                mov     ax, MSEC
                add     ax, 55             ;Interrupt every 55 msec.
                cmp     ax, 1000
                jb     SetMSEC

                inc     Timer             ;A second just passed.
                sub     ax, 1000          ;Adjust MSEC value.
SetMSEC:        mov     MSEC, ax

; End of critical region, restore the I flag to its former glory.

                popf

```

```

                pop     ax
                pop     ds
                jmp     cseg:OldInt1C;Transfer to original ISR.
TimerISR       endp

```

We will return to the problem of reentrancy and critical regions in the next two chapters of this text.

## 17.7 The Efficiency of an Interrupt Driven System

Interrupts introduce a considerable amount of complexity to a software system (see “Debugging ISRs” on page 1020). One might ask if using interrupts is really worth the trouble. The answer of course, is yes. Why else would people use interrupts if they were proven not to be worthwhile? However, interrupts are like many other nifty things in computer science – they have their place; if you attempt to use interrupts in an inappropriate fashion they will only make things worse for you.

The following sections explore the efficiency aspects of using interrupts. As you will soon discover, an interrupt driven system is usually superior despite the complexity. However, this is not always the case. For many systems, alternative methods provide better performance.

### 17.7.1 Interrupt Driven I/O vs. Polling

The whole purpose of an interrupt driven system is to allow the CPU to continue processing instructions while some I/O activity occurs. This is in direct contrast to a *polling system* where the CPU continually tests an I/O device to see if the I/O operation is complete. In an interrupt driven system, the CPU goes about its business and the I/O device interrupts it when it needs servicing. This is generally much more efficient than wasting CPU cycles polling a device while it is not ready.

The serial port is a perfect example of a device that works extremely well with interrupt driven I/O. You can start a communication program that begins downloading a file over a modem. Each time a character arrives, it generates an interrupt and the communication program starts up, buffers the character, and then returns from the interrupt. In the meantime, another program (like a word processor) can be running with almost no performance degradation since it takes so little time to process the serial port interrupts.

Contrast the above scenario with one where the serial communication program continually polls the serial communication chip to see if a character has arrived. In this case the CPU spends all of its time looking for an input character even though one rarely (in CPU terms) arrives. Therefore, no CPU cycles are left over to do other processing like running your word processor.

Suppose interrupts were not available and you wanted to allow background downloads while using your word processing program. Your word processing program would have to test the input data on the serial port once every few milliseconds to keep from losing any data. Can you imagine how difficult such a word processor would be to write? An interrupt system is the clear choice in this case.

If downloading data while word processing seems far fetched, consider a more simple case – the PC’s keyboard. Whenever a keypress interrupt occurs, the keyboard ISR reads the key pressed and saves it in the system type ahead buffer for the moment when the application wants to read the keyboard data. Can you imagine how difficult it would be to write applications if you had to constantly poll the keyboard port yourself to keep from losing characters? Even in the middle of a long calculation? Once again, interrupts provide an easy solution.

## 17.7.2 Interrupt Service Time

Of course, the serial communication system just described is an example of a *best case scenario*. The communication program takes so little time to do its job that most of the time is left over for the word processing program. However, were you to run a different interrupt driven I/O system, for example, copying files from one disk to another, the interrupt service routine would have a noticeable impact on the performance of the word processing system.

Two factors control an ISR's impact on a computer system: the *frequency of interrupts* and the *interrupt service time*. The frequency is how many times per second (or other time measurement) a particular interrupt occurs. The interrupt service time is how long the ISR takes to service the interrupt.

The nature of the frequency varies according to source of the interrupt. For example, the timer chip generates evenly spaced interrupts about 18 times per second, likewise, a serial port receiving at 9600bps generates better than 100 interrupts per second. On the other hand, the keyboard rarely generates more than about 20 interrupts per second and they are not very regular.

The interrupt service time is obviously dependent upon the number of instructions the ISR must execute. The interrupt service time is also dependent upon the particular CPU and clock frequency. The same ISR executing identical instructions on two CPUs will run in less time on a faster machine.

The amount of time an interrupt service routine takes to handle an interrupt, multiplied by the frequency of the interrupt, determines the impact the interrupt will have on system performance. Remember, every CPU cycle spent in an ISR is one less cycle available for your application programs. Consider the timer interrupt. Suppose the timer ISR takes 100  $\mu$ sec to complete its tasks. This means that the timer interrupt consumes 1.8 msec out of every second, or about 0.18% of the total computer time. Using a faster CPU will reduce this percentage (by reducing the time spent in the ISR); using a slower CPU will increase the percentage. Nevertheless, you can see that a short ISR such as this one will not have a significant effect on overall system performance.

One hundred microseconds is fast for a typical timer ISR, especially when your system has several timer ISRs chained together. However, even if the timer ISR took ten times as long to execute, it would only rob the system of less than 2% of the available CPU cycles. Even if it took 100 times longer (10 msec), there would only be an 18% performance degradation; most people would barely notice such a degradation<sup>9</sup>.

Of course, one cannot allow the ISR to take as much time as it wants. Since the timer interrupt occurs every 55 msec, the *maximum* time the ISR can use is just under 55msec. If the ISR requires more time than there is between interrupts, the system will eventually lose an interrupt. Furthermore, the system will spend all its time servicing the interrupt rather than accomplishing anything else.

For many systems, having an ISR that consumes as much as 10% of the overall CPU cycles will not prove to a problem. However, before you go off and start designing slow interrupt service routines, you should remember that your ISR is probably not the only ISR in the system. While your ISR is consuming 25% of the CPU cycles, there may be another ISR that is doing the same thing; and another, and another, and... Furthermore, there may be some ISRs that require fast servicing. For example, a serial port ISR may need to read a character from the serial communications chip each millisecond or so. If your timer ISR requires 4 msec to execute and does so with the interrupts turned off, the serial port ISR will miss some characters.

Ultimately, of course, you would like to write ISRs so they are as fast as possible so they have as little impact on system performance as they can. This is one of the main rea-

---

9. As a general rule, people begin to notice a real difference in performance between 25 and 50%. It isn't instantly obvious until about 50% (i.e., running at one-half the speed).

sons most ISRs for DOS are still written in assembly language. Unless you are designing an *embedded system*, one in which the PC runs only your application, you need to realize that your ISRs must coexist with other ISRs and applications; you do not want the performance of your ISR to adversely affect the performance of other code in the system.

### 17.7.3 Interrupt Latency

Interrupt latency is the time between the point a device signals that it needs service and the point where the ISR provides the needed service. This is not instantaneous! At the very least, the 8259 PIC needs to signal the CPU, the CPU needs to interrupt the current program, push the flags and return address, obtain the ISR address, and transfer control to the ISR. The ISR may need to push various registers, set up certain variables, check device status to determine the source of the interrupt, and so on. Furthermore, there may be other ISRs chained into the interrupt vector before you and they execute to completion before transferring control to your ISR that actually services the device. Eventually, the ISR actually does whatever it is that the device needs done. In the best case on the fastest microprocessors with simple ISRs, the latency could be under a microsecond. On slower systems, with several ISRs in a chain, the latency could be as bad as several *milliseconds*.

For some devices, the interrupt latency is more important than the actual interrupt service time. For example, an input device may only interrupt the CPU once every 10 seconds. However, that device may be incapable of holding the data on its input port for more than a millisecond. In theory, any interrupt service time less than 10 seconds is fine; but the CPU must read the data within one millisecond of its arrival or the system will lose the data.

Low interrupt latency (that is, responding quickly) is very important in many applications. Indeed, in some applications the latency requirements are so strict that you have to use a very fast CPU or you have to abandon interrupts altogether and go back to polling. *What a minute!* Isn't polling less efficient than an interrupt driven system? How will polling improve things?

An interrupt driven I/O system improves system performance by allowing the CPU to work on other tasks in between I/O operations. In principle, servicing interrupts takes very little CPU time compared the arrival of interrupts to the system. By using interrupt driven I/O, you can use all those other CPU cycles for some other purpose. However, suppose the I/O device is producing service requests at such a rate that there are no free CPU cycles. Interrupt driven I/O will provide few benefits in this case.

For example, suppose we have an eight bit I/O device connected to two I/O ports. Suppose bit zero of port 310h contains a one if data is available and a zero otherwise. If data is available, the CPU must read the eight bits at port 311h. Reading port 311h clears bit zero of port 310h until the next byte arrives. If you wanted to read 8192 bytes from this port, you could do this with the following short segment of code:

```

                                mov     cx, 8192
                                mov     dx, 310h
                                lea     bx, Array                ;Point bx at storage buffer
DataAvailLp:                    in      al, dx                 ;Read status port.
                                shr     al, 1                  ;Test bit zero.
                                jnc     DataAvailLp            ;Wait until data is
available.
                                inc     dx                     ;Point at data port.
                                in      al, dx                 ;Read data.
                                mov     [bx], al              ;Store data into buffer.
                                inc     bx                     ;Move on to next array
element.
                                dec     dx                     ;Point back at status port.
                                loop    DataAvailLp           ;Repeat 8192 times.
                                .
                                .

```

This code uses a classical polling loop (`DataAvailP`) to wait for each available character. Since there are only three instructions in the polling loop, this loop can probably execute in just under a microsecond<sup>10</sup>. So it might take as much as one microsecond to determine that data is available, in which case the code falls through and by the second instruction in the sequence we've read the data from the device. Let's be generous and say that takes another microsecond. Suppose, instead, we use an interrupt service routine. A *well-written* ISR combined with a good system hardware design will probably have latencies measured in microseconds.

To measure the *best case* latency we could hope to achieve would require some sort of hardware timer than begins counting once an interrupt event occurs. Upon entry into our interrupt service routine we could read this counter to determine how much time has passed between the interrupt and its service. Fortunately, just such a device exists on the PC – the 8254 timer chip that provides the source of the 55 msec interrupt.

The 8254 timer chip actually contains three separate timers: timer #0, timer #1, and timer #2. The first timer (timer #0) provides the clock interrupt, so it will be the focus of our discussion. The timer contains a 16 bit register that the 8254 decrements at regular intervals (1,193,180 times per second). Once the timer hits zero, it generates an interrupt on the 8259 IRQ 0 line and then wraps around to 0FFFFh and continues counting down from that point. Since the counter automatically resets to 0FFFFh after generating each interrupt, this means that the 8254 timer generates interrupts every 65,536/1,193,180 seconds, or once every 54.9254932198 msec, which is 18.2064819336 times per second. We'll just call these once every 55 msec or 18 (or 18.2) times per second, respectively. Another way to view this is that the 8254 decrements the counter once every 838 nanoseconds (or 0.838  $\mu$ sec).

The following short assembly language program measures interrupt latency by patching into the int 8 vector. Whenever the timer chip counts down to zero, it generates an interrupt that directly calls this program's ISR. The ISR quickly reads the timer chip's counter register, negates the value (so 0FFFFh becomes one, 0FFFEh becomes two, etc.), and then adds it to a running total. The ISR also increments a counter so that it can keep track of the number of times it has added a counter value to the total. Then the ISR jumps to the original int 8 handler. The main program, in the mean time, simply computes and displays the current average read from the counter. When the user presses any key, this program terminates.

```
; This program measures the latency of an INT 08 ISR.
; It works by reading the timer chip immediately upon entering
; the INT 08 ISR. By averaging this value for some number of
; executions, we can determine the average latency for this
; code.

                                .xlist
                                .386
                                option      segment:use16
                                include      stdlib.a
                                includelib  stdlib.lib
                                .list

cseg                             segment  para public 'code'
                                assume    cs:cseg, ds:nothing

; All the variables are in the code segment in order to reduce ISR
; latency (we don't have to push and set up DS, saving a few instructions
; at the beginning of the ISR).

OldInt8                          dword   ?
SumLatency                       dword   0
```

---

10. On a fast CPU (e.g. 100 MHz Pentium), you might expect this loop to execute in much less time than one microsecond. However, the in instruction is probably going to be quite slow because of the wait states associated with external I/O devices.

```

Executions      dword    0
Average         dword    0

; This program reads the 8254 timer chip. This chip counts from
; 0FFFFh down to zero and then generates an interrupt. It wraps
; around from 0 to 0FFFFh and continues counting down once it
; generates the interrupt.
;
; 8254 Timer Chip port addresses:

Timer0_8254     equ      40h
Cntrl_8254      equ      43h

; The following ISR reads the 8254 timer chip, negates the result
; (because the timer counts backwards), adds the result to the
; SumLatency variable, and then increments the Executions variable
; that counts the number of times we execute this code. In the
; mean time, the main program is busy computing and displaying the
; average latency time for this ISR.
;
; To read the 16 bit 8254 counter value, this code needs to
; write a zero to the 8254 control port and then read the
; timer port twice (reads the L.O. then H.O. bytes). There
; needs to be a short delay between reading the two bytes
; from the same port address.

TimerISR        proc      near
                push     ax
                mov      eax, 0           ;Ch 0, latch & read data.
                out      Cntrl_8254, al  ;Output to 8253 cmd register.
                in       al, Timer0_8254 ;Read latch #0 (LSB) & ignore.
                mov      ah, al
                jmp      SettleDelay     ;Settling delay for 8254 chip.
SettleDelay:    in       al, Timer0_8254 ;Read latch #0 (MSB)
                xchg     ah, al
                neg      ax             ;Fix, 'cause timer counts down.
                add     cseg:SumLatency, eax
                inc     cseg:Executions
                pop      ax
                jmp     cseg:OldInt8
TimerISR        endp

Main            proc      meminit

; Begin by patching in the address of our ISR into int 8's vector.
; Note that we must turn off the interrupts while actually patching
; the interrupt vector and we must ensure that interrupts are turned
; back on afterwards; hence the cli and sti instructions. These are
; required because a timer interrupt could come along between the two
; instructions that write to the int 8 interrupt vector. Since the
; interrupt vector is in an inconsistent state at that point, this
; could cause the system to crash.

                mov     ax, 0
                mov     es, ax
                mov     ax, es:[8*4]
                mov     word ptr OldInt8, ax
                mov     ax, es:[8*4 + 2]
                mov     word ptr OldInt8+2, ax

                cli
                mov     word ptr es:[8*4], offset TimerISR
                mov     es:[8*4 + 2], cs
                sti

; First, wait for the first call to the ISR above. Since we will be dividing

```



```
; by the value in the Executions variable, we need to make sure that it is
; greater than zero before we do anything.
```

```
Wait4Non0:      cmp      cseg:Executions, 0
                je       Wait4Non0
```

```
; Okay, start displaying the good values until the user presses a key at
; the keyboard to stop everything:
```

```
DisplayLp:     mov      eax, SumLatency
                cdq
                div     Executions           ;Extends eax->edx.
                mov     Average, eax
                printf
                byte   "Count: %ld, average: %ld\n",0
                dword  Executions, Average

                mov     ah, 1                ;Test for keystroke.
                int    16h
                je     DisplayLp
                mov     ah, 0                ;Read that keystroke.
                int    16h
```

```
; Okay, restore the interrupt vector. We need the interrupts off
; here for the same reason as above.
```

```
                mov     ax, 0
                mov     es, ax
                cli
                mov     ax, word ptr OldInt8
                mov     es:[8*4], ax
                mov     ax, word ptr OldInt8+2
                mov     es:[8*4+2], ax
                sti

Quit:           ExitPgm                       ;DOS macro to quit program.
Main           endp

cseg           ends

sseg          segment para stack 'stack'
stk           db      1024 dup ("stack ")
sseg          ends

zzzzzzseg     segment para public 'zzzzzz'
LastBytes    db      16 dup (?)
zzzzzzseg     ends
end           Main
```

On a 66 MHz 80486 DX/2 processor, the above code reports an average value of 44 after it has run for about 10,000 iterations. This works out to about 37  $\mu$ sec between the device signalling the interrupt and the ISR being able to process it<sup>11</sup>. *The latency of polled I/O would probably be an order of magnitude less than this!*

Generally, if you have some high speed application like audio or video recording or playback, you probably cannot afford the latencies associated with interrupt I/O. On the other hand, such applications demand such high performance out of the system, that you probably wouldn't have any CPU cycles left over to do other processing while waiting for I/O.

---

11. Patching into the int 1Ch interrupt vector produces latencies in the 137  $\mu$ sec range.

Another issue with respect to ISR latency is *latency consistency*. That is, is there the same amount of latency from interrupt to interrupt? Some ISRs can tolerate considerable latency as long as it is consistent (that is, the latency is roughly the same from interrupt to interrupt). For example, suppose you want to patch into the timer interrupt so you can read an input port every 55 msec and store this data away. Later, when processing the data, your code might work under the assumption that the data readings are 55 msec (or 54.9...) apart. This might not be true if there are other ISRs in the timer interrupt chain before your ISR. For example, there may be an ISR that counts off 18 interrupts and then executes some code sequence that requires 10 msec. This means that 16 out of every 18 interrupts your data collection routine would collect data at 55 msec intervals right on the nose. But when that 18<sup>th</sup> interrupt occurs, the other timer ISR will delay 10 msec before passing control to your routine. This means that your 17<sup>th</sup> reading will be 65 msec since the last reading. Don't forget, the timer chip is still counting down during all of this, that means there are now only 45 msec to the next interrupt. Therefore, your 18<sup>th</sup> reading would occur 45 msec after the 17<sup>th</sup>. Hardly a consistent pattern. If your ISR needs a consistent latencies, you should try to install your ISR as early in the interrupt chain as possible.

#### 17.7.4 Prioritized Interrupts

Suppose you have the interrupts turned off for a brief spell (perhaps you are processing some interrupt) and *two* interrupt requests come in while the interrupts are off. What happens when you turn the interrupts back on? Which interrupt will the CPU first service? The obvious answer would be "whichever interrupt occurred first." However, suppose the both occurred at exactly the same time (or, at least, within a short enough time frame that we cannot determine which occurred first), or maybe, as is really the case, the 8259 PIC cannot keep track of which interrupt occurred first? Furthermore, what if one interrupt is more important than another? Suppose for example, that one interrupt tells that the user has just pressed a key on the keyboard and a second interrupt tells you that your nuclear reactor is about to melt down if you don't do something in the next 100 µsec. Would you want to process the keystroke first, even if its interrupt came in first? Probably not. Instead, you would want to *prioritize* the interrupts on the basis of their importance; the nuclear reactor interrupt is probably a little more important than the keystroke interrupt, you should probably handle it first.

The 8259 PIC provides several priority schemes, but the PC BIOS initializes the 8259 to use *fixed* priority. When using fixed priorities, the device on IRQ 0 (the timer) has the highest priority and the device on IRQ 7 has the lowest priority. Therefore, the 8259 in the PC (running DOS) always resolves conflicts in this manner. If you *were* going to hook that nuclear reactor up to your PC, you'd probably want to use the *nonmaskable* interrupt since it has a higher priority than anything provided by the 8259 (and you can't mask it with a CLI instruction).

### 17.8 Debugging ISRs

Although writing ISRs can simplify the design of many types of programs, ISRs are almost always very difficult to debug. There are two main reasons ISRs are more difficult than standard applications to debug. First, as mentioned earlier, errant ISRs can modify values the main program uses (or, worse yet, that some *other* program in memory is using) and it is difficult to pin down the source of the error. Second, most debuggers have fits when you attempt to set breakpoints within an ISR.

If your code includes some ISRs and the program seems to be misbehaving and you cannot immediately see the reason, you should immediately suspect interference by the ISR. Many programmers have forgotten about ISRs appearing in their code and have spent weeks attempting to locate a bug in their non-ISR code, only to discover the problem was with the ISR. Always suspect the ISR first. Generally, ISRs are short and you can

quickly eliminate the ISR as the cause of your problem before trying to track the bug down elsewhere.

Debuggers often have problems because they are not reentrant or they call BIOS or DOS (that are not reentrant) so if you set a breakpoint in an ISR that has interrupted BIOS or DOS and the debugger calls BIOS or DOS, the system may crash because of the reentrancy problems. Fortunately, most modern debuggers have a *remote* debugging mode that lets you connect a terminal or another PC to a serial port and execute the debug commands on that second display and keyboard. Since the debugger talks directly to the serial chip, it avoids calling BIOS or DOS and avoids the reentrancy problems. Of course, this doesn't help much if you're writing a *serial* ISR, but it works fine with most other programs.

A big problem when debugging interrupt service routines is that the system crashes immediately after you patch the interrupt vector. If you do not have a remote debugging facility, the best approach to debug this code is to strip the ISR to its bare essentials. This might be the code that simply passes control on to the next ISR in the interrupt chain (if applicable). Then add one section of code at a time back to your ISR until the ISR fails.

Of course, the best debugging strategy is to write code that doesn't have any bugs. While this is not a practical solution, one thing you can do is attempt to do as little as possible in the ISR. Simply read or write the device's data and buffer any inputs for the main program to handle later. The smaller your ISR is, the less complex it is, the higher the probability is that it will not contain any bugs.

Debugging ISRs, unfortunately, is not easy and it is not something you can learn right out of a book. It takes lots of experience and you will need to make a lot of mistakes. There is unfortunately, but there is no substitute for experience when debugging ISRs.

## 17.9 Summary

This chapter discusses three phenomena occurring in PC systems: interrupts (hardware), traps, and exceptions. An interrupt is an asynchronous procedure call the CPU generates in response to an external hardware signal. A trap is a programmer-supplied call to a routine and is a special form of a procedure call. An exception occurs when a program executes an instruction that generates some sort of error. For additional details, see

- “Interrupts, Traps, and Exceptions” on page 995.

When an interrupt, trap, or exception occurs, the 80x86 CPU pushes the flags and transfers control to an *interrupt service routine* (ISR). The 80x86 supports an *interrupt vector table* that provides segmented addresses for up to 256 different interrupts. When writing your own ISR, you need to store the address of your ISR in an appropriate location in the interrupt vector table to activate that ISR. Well-behaved programs also save the original interrupt vector value so they can restore it when they terminate. For the details, see

- “80x86 Interrupt Structure and Interrupt Service Routines (ISRs)” on page 996

A *trap*, or *software interrupt*, is nothing more than the execution of an 80x86 “int n” instruction. Such an instruction transfers control to the ISR whose vector appears in the n<sup>th</sup> entry in the interrupt vector table. Generally, you would use a trap to call a routine in a resident program appearing somewhere in memory (like DOS or BIOS). For more information, see

- “Traps” on page 999

An exception occurs whenever the CPU executes an instruction and that instruction is illegal or the execution of that instruction generates some sort of error (like division by zero). The 80x86 provides several built-in exceptions, although this text only deals with the exceptions available in real mode. For the details, see

- “Exceptions” on page 1000

- “Divide Error Exception (INT 0)” on page 1000
- “Single Step (Trace) Exception (INT 1)” on page 1000
- “Breakpoint Exception (INT 3)” on page 1001
- “Overflow Exception (INT 4/INTO)” on page 1001
- “Bounds Exception (INT 5/BOUND)” on page 1001
- “Invalid Opcode Exception (INT 6)” on page 1004
- “Coprocessor Not Available (INT 7)” on page 1004

The PC provides hardware support for up to 15 vectored interrupts using a pair of 8259A programmable interrupt controller chips (PICs). Devices that normally generate hardware interrupts include a timer, the keyboard, serial ports, parallel ports, disk drives, sound cards, the real time clock, and the FPU. The 80x86 lets you enable and disable all *maskable* interrupts with the `cli` and `sti` instructions. The PIC also lets you individually mask the devices that can interrupt the system. However, the 80x86 provides a special *nonmaskable* interrupt that has a higher priority than the other hardware interrupts and cannot be disabled by a program. For more details on these hardware interrupts, see

- “Hardware Interrupts” on page 1004
- “The 8259A Programmable Interrupt Controller (PIC)” on page 1005
- “The Timer Interrupt (INT 8)” on page 1007
- “The Keyboard Interrupt (INT 9)” on page 1008
- “The Serial Port Interrupts (INT 0Bh and INT 0Ch)” on page 1008
- “The Parallel Port Interrupts (INT 0Dh and INT 0Fh)” on page 1008
- “The Diskette and Hard Drive Interrupts (INT 0Eh and INT 76h)” on page 1009
- “The Real-Time Clock Interrupt (INT 70h)” on page 1009
- “The FPU Interrupt (INT 75h)” on page 1009
- “Nonmaskable Interrupts (INT 2)” on page 1009
- “Other Interrupts” on page 1009

Interrupt service routines that you write may need to coexist with other ISRs in memory. In particular, you may not be able to simply replace an interrupt vector with the address of your ISR and let your ISR take over from there. Often, you will need to create an *interrupt chain* and call the previous ISR in the interrupt chain once you are done processing the interrupt. To see why you create interrupt chains, and to learn how to create them, see

- “Chaining Interrupt Service Routines” on page 1010

With interrupts comes the possibility of *reentrancy*, that is, the possibility that a routine might be interrupt and called again before the first call finished execution. This chapter introduces the concept of reentrancy and gives some examples that demonstrate problems with nonreentrant code. For details, see

- “Reentrancy Problems” on page 1012

The whole purpose of an interrupt driven system is to improve the efficiency of that system. Therefore, it should come as no surprise that ISRs should be as efficient as possible. This chapter discusses why interrupt driven I/O systems can be more efficient and contrasts interrupt driven I/O with *polled* I/O. However, interrupts can cause problems if the corresponding ISR is too slow. Therefore, programmers who write ISRs need to be aware of such parameters as *interrupt service time*, *frequency of interrupts*, and *interrupt latency*. To learn about these concepts, see

- “The Efficiency of an Interrupt Driven System” on page 1014
- “Interrupt Driven I/O vs. Polling” on page 1014
- “Interrupt Service Time” on page 1015
- “Interrupt Latency” on page 1016

If multiple interrupts occur simultaneously, the CPU must decide which interrupt to handle first. The 8259 PIC and the PC use a prioritized interrupt scheme assigning the highest priority to the timer and work down from there. The 80x86 always processes the interrupt with the highest priority first. For more details, see

- “Prioritized Interrupts” on page 1020

