At this writing, PCI appears to be a very strong contender for becoming the primary bus system of PCs in general and is the architecture of choice for current high end, mainstream PCs. Its 264 MB/sec peak data transfer rate is high enough to handle the memory bandwidth requirements of even the highest speed processors, and it has many advanced features such as processor independence, hidden bus arbitration, multiple bus master support, and long length burst mode transfers. PCI is discussed further in Section 7-9.

## 7-4. Interrupt Controller

No modern computer system could handle the demands imposed on it by the multiple I/O devices it supports without an ability to support *hardware interrupts.* A simple way to appreciate how hardware interrupts work and their importance is by analogy with the telephone system and its role in your own life. When people in other places want to send or receive information from you, they can do it by the telephone system. Think of the phone system's wires as being like the wires connecting an I/O device like the keyboard to the CPU, and you have an appropriate context in which to understand interrupts. A computer system without interrupts is like a telephone without a ringer. If your phone had no ringer, you would need to pick up the phone every few seconds to see if someone was trying to call you. Similarly, if your computer system had no hardware interrupts, the CPU would have to spend most of its time going around checking to see if any I/O devices had new data. With a phone ringer, you can just go about your business until the phone rings and only then turn your attention to whomever is trying to reach you. An interrupt serves as the telephone ringer for the CPU. The CPU can go about doing various tasks without worrying about the devices it's connected to until an interrupt occurs. Then and only then it tends to the needs of the device. The hardware interrupt concept is so important that much of Chapter 8 is devoted to discussing interrupt applications.

The analogy can be taken a step further. Suppose you're reading a book and the phone rings. What you do is to mark the place where you were in your book and then answer the phone. When the phone call is over, you use the book mark to find out where you were, and then resume reading again. The CPU's response to an interrupt is much the same. It saves its place in the code it's currently executing by pushing all essential information about its current state (such as cs:ip and the flags) on the stack, and then it jumps to a routine that services the interrupt. This routine is known as an interrupt handler, or alternatively, as an interrupt service routine (ISR). When the handler has finished executing, the CPU pops the information it stored on the stack and continues executing the previous code as if nothing had ever happened.

As mentioned in Section 7-1, the CPU has only a single interrupt line to let it know that some device needs service. But typically there are a number of devices that need the

CPU's attention from to time, and there needs to be some way of coordinating their requests for attention. This is where the *interrupt controller* comes in. It acts like an executive's secretary, screening multiple phone calls and only putting them through to the executive one at a time in the order of their importance. In the same way, the interrupt controller has interrupt request lines from a number of I/O devices connected to it, and it passes those requests on to the CPU via its interrupt line one at a time in a prioritized order of importance. When the CPU receives an interrupt, it checks with the interrupt controller (after saving its place in whatever task it's executing) to find out which device caused that interrupt and then jumps to the appropriate interrupt handler.

## 8259A Interrupt Controller

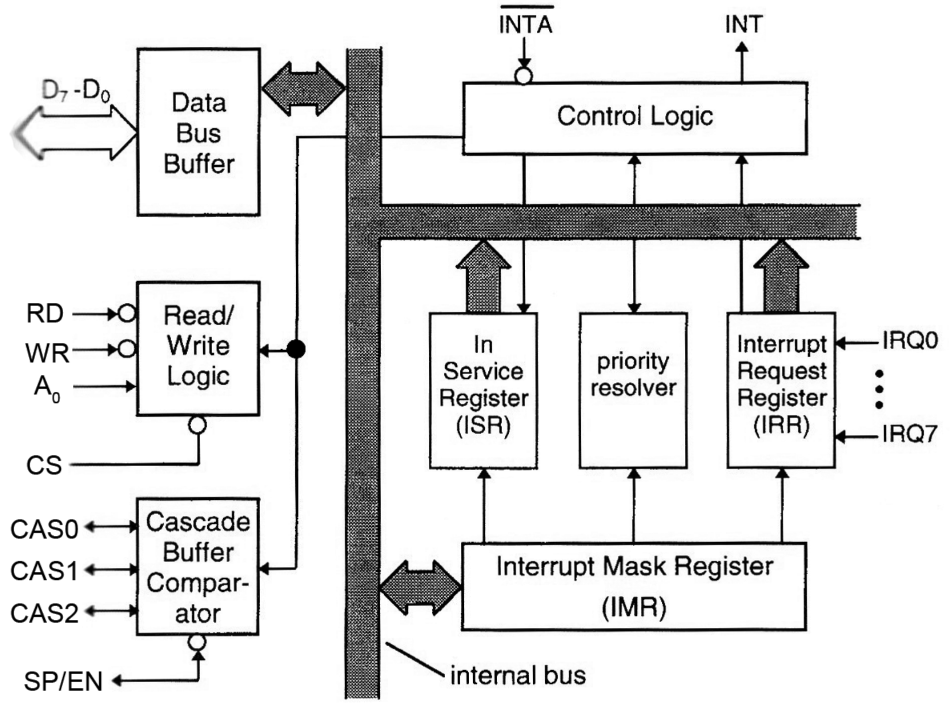
The original IBM PC used an Intel 8259A programmable interrupt controller (PIC) to handle multiple interrupt sources. In the PC-AT, the interrupt-handling capabilities were expanded by adding a second 8259A. In order to remain compatible, all PCs since the PC-AT have incorporated circuitry that's functionally equivalent to a pair 8259As on their motherboards, although the circuitry normally appears somewhere inside a motherboard chip set rather than as discrete 8259As. Thus you can count on a PC to behave as if it had two 8259A chips on its motherboard even though you won't see them there. With this in mind, let's look at how the 8259A interrupt controller works and how a pair of them are coordinated in a PC.

Figure 7-12 shows a block diagram of an 8259A. It's a programmable device whose internal registers can be read and written by the CPU through a pair of I/O ports. While different options and modes of operation are available by programming the registers, we only describe the behavior of the 8259A for the standard operating configuration used in PCs. For further information about the 8259A's many programmable features, see the data sheet for the chip in Intel's *Component Data Catalog* and van Gilluwe (1994).

The 8259A supports interrupt request lines coming from up to eight different devices. These lines are labeled as IRQ*n*, where *n* = 0-7. A external device requests an interrupt by producing a rising edge (a low to high logic level transition) on its IRQ line. The 8259A prioritizes interrupt requests by their IRQ number, with, by default, IRQ0 being the highest priority and IRQ7 the lowest. Thus if two or more IRQs come in at the same time, the higher priority request is recognized and sent on to the CPU while the lower priority request has to wait. In addition, if an IRQ comes in and there's already another interrupt of lower priority being serviced, the interrupt request is recognized by the 8259A and passed on to the CPU. It can then interrupt its handling of the lower priority interrupt that's in progress.

You can prevent any interrupt request by setting bits in the 8259A's interrupt-mask register. This register contains a byte whose *n*th bit *masks* the interrupt request from line IRQ*n*. That is, if bit *n* is nonzero, line IRQ*n* cannot cause an interrupt. When the 8259A recognizes an interrupt request and passes it on to the CPU, it does so by setting its interrupt line (INT) high. This line is directly connected to the CPU's interrupt input line (INTR). If the CPU's interrupts are enabled (by prior execution of an sti instruction), the CPU acknowledges the interrupt by sending an interrupt acknowledge signal (a pair of active low pulses on the 8259A's $\overline{\text{INTA}}$ line, generated by external logic when the CPU's W/$\overline{\text{R}}$, M/$\overline{\text{IO}}$, and D/$\overline{\text{C}}$ signals are all low) back to the 8259A. The 8259A responds to the $\overline{\text{INTA}}$ signal by pulling INTR back low, and then putting an 8-bit interrupt type code *nn* onto the data lines. The type code tells the CPU which interrupt vector to select in the interrupt vector table. In real mode, this table consists of four-byte entries and begins at memory location 0:0. More specifically, if the 8259A places the number *nn* on the bus, then the interrupt vector at absolute address 0:4**nn* is used. Alternatively, if the CPU is operating in protected mode, the interrupt vector at offset 8**nn* in the Interrupt Descriptor Table (IDT—see Section 5-2) is used. The CPU then pushes its flags onto the stack and executes a far call to the address given by this interrupt vector. You may recall that this is exactly the way the software int *nn* instruction discussed in Section 3-6 works. The appropriate interrupt handler program (see Section 4-6) had better be present at the called address, or there'll be a crash! In essence, a hardware interrupt can be described as an asynchronous call to some memory location caused by an external device.

**FIGURE 7-12.** Block diagram of the Intel 8259A programmable interrupt controller, which coordinates the interrupt requests appearing on its eight interrupt request lines (IRQ*n*).



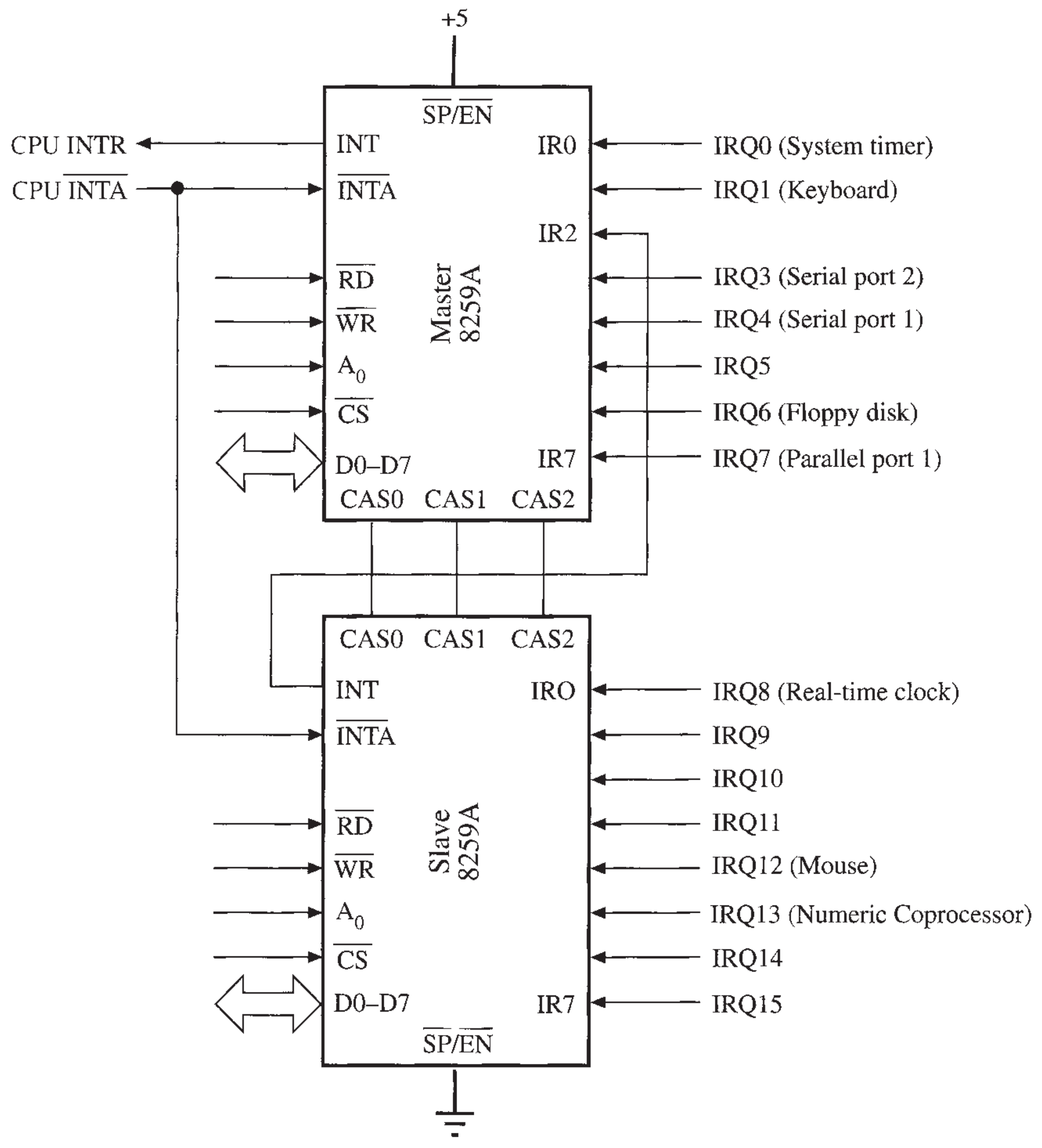
Before discussing the software required to initialize the 8259A appropriately and to handle interrupt requests, let's look further at the 8259A's hardware and the signals it uses. As can be seen in Figure 7-12, the 8259A signals that the CPU uses to program the 8259A's internal registers are the typical signals used by many smart peripheral chips (see Section 7-2). Although the 8259A has a number of internal registers, only two I/O ports are used to access them (see code below for how this is done). To communicate through the ports, the 8259A uses its I/O read ($\overline{\text{IOR}}$), I/O write ($\overline{\text{IOW}}$), chip-select ($\overline{\text{CS}}$), data (D0-D7), and address line A0 (to select one of the two I/O ports). The interrupt (INT) output line, the interrupt acknowledge line ($\overline{\text{INTA}}$) and the eight interrupt request lines (IRQ*n*) have already been discussed above. This leaves only the cascade lines CAS0, CAS1, and CAS2 plus the slave program/enable buffer line ($\overline{\text{SP}}/\overline{\text{EN}}$). These four lines are used to coordinate two or more 8259As so that they can work together in a single system. As mentioned earlier, there are in fact the equivalent of a coordinated pair of 8259As on all PC motherboards.

The way the 8259As are connected is shown in Figure 7-13. The $\overline{\text{SP}}/\overline{\text{EN}}$ line tells the 8259A it's a master controller if the line is high, and a slave controller if it's low. The three cascade lines are then used by the master to tell the slave when an interrupt acknowledge signal sent by the CPU is meant for it.

In the PC, the IRQ lines for the controller pair are numbered from 0 to 15, with IRQ0-IRQ7 being on the master 8259A and IRQ8-IRQ15 being on the slave. Notice that IRQ2 is used by the slave controller to relay its INT signal back to the master, leaving IRQ2 unavailable for use by external devices. This leaves 15 usable interrupt request lines. However, four of these are dedicated to specific systems devices (IRQ0 = system timer, IRQ1 = keyboard, IRQ8 = real-time clock, and IRQ13 = numeric coprocessor), leaving 11 IRQ lines that are brought out to the PC's ISA bus for use by devices on plug-in cards. A number of these have more or less standardized uses as well, including IRQ4 = serial port 1, IRQ6 = floppy disk controller, IRQ7 = parallel port 1, and IRQ12 = mouse interface). In addition, IRQ3 is normally used for the second serial port if it's present, and IRQ5 is used by many network interface cards.

It's important to note that the connection of the slave 8259A through IRQ2 on the master rearranges the priorities of the interrupt requests. Although the highest priority IRQ within each 8259A is still IRQ0 for the master and IRQ8 for the slave, the chip connection through IRQ2 causes IRQ8-IRQ15 to be the next highest priority requests after IRQ1. Thus the default priority order for interrupt requests is IRQ0, IRQ1, IRQ8-IRQ15, IRQ3-IRQ7.

**FIGURE 7-13.** The equivalent of two 8259A interrupt controllers connected together as shown are present in all PCs. Some standard PC assignments for the interrupt request lines are also shown.



## Interrupt Controller Programming

To initialize the operating mode for the two 8259As in a PC, four initialization command words (ICWs) must be sent to each of them. The first ICW is sent to the 8259A's first port address (20h or 0a0h), and the remaining ICWs are sent the

8259A's second port address (21h or 0a1h). The proper initialization can be done with the code

```
MASTR0  equ    20h            ;Port addresses for first (master) 8259A
MASTR1  equ    21h
SLAVE0  equ    0a0h           ;Port addresses for second (slave) 8259A
SLAVE1  equ    0a1h

        mov    al,11h         ;ICW1: set edge-triggered IRQs, cascade
        out    MASTR0,al      ; mode, ICW4 is needed
        mov    al,8           ;ICW2: use int vectors 8-0fh for IRQ0-
        out    MASTR1,al      ; IRQ7: conflicts with x86 exceptions
        mov    al,4           ;ICW3: slave 8259A is on IRQ2
        out    MASTR1,al
        mov    al,1           ;ICW4: non-buffered mode, normal EOI,
        out    MASTR1,al      ; x86 mode
        mov    al,11h         ;ICW1: set edge-triggered IRQs, cascade
        out    SLAVE0,al      ; mode, ICW4 is needed
        mov    al,70h         ;ICW2: use int vectors 70-7Fh for IRQ8-
        out    SLAVE1,al      ; IRQ15
        mov    al,2           ;ICW3: slave 8259A ID is 2
        out    SLAVE1,al
        mov    al,1           ;ICW4: non-buffered mode, normal EOI,
        out    SLAVE1,al      ; x86 mode
```

This sets up the 8259As to respond to positive-going edges on the interrupt request lines (IRQ*n*), to use interrupt vectors 8-0fh or 70h-7fh (see Table 3-4 for interrupt-vector assignments), to work with an x86 processor (rather than an 8080/8085), and to reinitialize interrupts upon receipt of an end-of-interrupt (EOI) code. At this point the 8259As are ready to accept interrupt requests on their IRQ lines, and to raise the INT output accordingly. All subsequent writes to port 21h or 0a1h after the four ICWs have been sent access the interrupt-mask register.

As discussed in Section 5-5, the PC default use of interrupt vectors 8 - 0fh for the master 8259A conflicts with their use for very important x86 exceptions, such as the General Protection exception, which uses interrupt vector 0dh. To get around such conflicts, standard protected-mode operating systems such as Microsoft Windows program the master 8259A to use interrupt vectors 50h through 57h instead of 8 through 0fh.

Interrupts are enabled on the CPU by execution of the set-interrupt-flag (sti) instruction. Before the PC BIOS routines do this during the PC's power-on sequence, they mask off (disable) all but four of the IRQ*n* lines. Specifically the instructions

```
        mov    al,0b8h        ;Enable disk (bit 6), slave 8259A (2),
        out    MASTR1,al      ; keyboard (1),and timer (0) interrupts
        mov     al,0ffh         ;disable all slave 8259A IRQs
        out     SLAVE1,al
        sti
```

are executed (after some initial testing of the 8259As is performed). When an interrupt occurs, further interrupts are disabled in the CPU, and each 8259A records but does not request further service until it receives an EOI code. Thus it is the responsibility of all interrupt handler programs to reenable interrupts by executing the sequence

```
        mov     al,20h          ;EOI command
        out     MASTR0,al       ;Replace MASTR0 with SLAVE0 for slave
        sti                     ; 8259A
```

For examples of timer and keyboard interrupt routines, see Sections 7-5, 8-1, and 8-2. For an example of a serial-port interrupt routine, see Section 11-5.

The removal of IRQ2 for cascading the 8259As in the PC-AT created a compatibility problem with previous PCs, which had IRQ2 brought out to the ISA bus on pin B4. To solve this problem, the IRQ9 line was connected to this bus pin instead, and interrupt requests coming in on IRQ9 are redirected so they use the interrupt vector that would have been used by IRQ2, which is the vector for int 0ah if the 8259A is initialized as above. This can done by having the interrupt handler corresponding to IRQ9, that is, the handler for int 71h, include the simple code

```
intseg  segment at 0
        org     4*0ah
intIRQ2 label   far             ;Default offset of IRQ2 handler
intseg  ends

        push    ax
        mov     al,20h          ;Send EOI signal to 8259A for IRQ9
        out     SLAVE0,al
        pop     ax
        jmp     intIRQ2         ;Chain to IRQ2 handler
```

You can connect your own sources of interrupt requests to IRQ3-IRQ5, IRQ7, IRQ9-IRQ12, and IRQ14-IRQ15 as these interrupt request lines are all connected to pins on the ISA bus for use by plug-in boards. Keep in mind, however, that there are semi-standardized assignments for all of these interrupt request lines except IRQ9-IRQ11 and IRQ15 (see Table 3-5). It's sometimes tempting to try to attach more than one device to a given IRQ line, but IBM's original choice of programming the 8259A to respond only to rising edges on the IRQ lines makes this a bit tricky. To make sharing work, the devices sharing an IRQ must each have a status register containing an "interrupt pending" bit that the CPU can read to see which device(s) have requested an interrupt. Usually, reading this bit resets the bit. The difficulties occur because in the process of servicing an interrupt for one device, another on the same IRQ channel may try to interrupt, only to have its edge trigger ignored so long as interrupts are disabled. To take care of such attempted interrupts, before returning from an interrupt, the interrupt handler needs to reestablish interrupts and then poll all possible devices that share the IRQ channel, servicing any that request interrupts.

To eliminate the need for such polling, the IBM PS/2s program the 8259As to use *level-triggered* interrupts. Specifically, the IRQ*n* lines are driven by open-collector circuits, which allow more than one interrupting device to hold an IRQ*n* line low (see the "wired-OR" discussion in Section 6-3). When interrupt service completes for a device on IRQ*n*, the 8259A interrupts again as long as any device holds that IRQ*n* line low. It's crucial that each interrupt handler clear the device flip-flop that's responsible for pulling the IRQ*n* line low *before* fully reenabling interrupts, since otherwise an infinite interrupt loop occurs, which may be terminated by a stack overflow. If you find that your code works fine on regular PCs but crashes on PS/2s, suspect this problem first. The CMOS RTC routine clock in Section 8-2 illustrates how an interrupt handler can be written to work with both edge-triggered and level-triggered interrupts.

To end this section, we summarize by giving a step-by-step description of all the actions that take place when a hardware interrupt occurs:

1. The requesting device generates a rising edge on its interrupt request line, IRQ*n*.
2. The 8259A checks its internal interrupt-mask register and in-service register. If interrupt level *n* is not masked off *and* if another interrupt of the same or higher priority is not in progress, the 8259A puts a logic high on the CPU's INTR line.
3. When its INTR line goes high, the CPU checks its internal interrupt flag (IF). If the CPU finds that interrupts are enabled (IF=1), the CPU responds to the interrupt. If interrupts are disabled (IF=0), the INTR line is ignored.
4. If interrupts were enabled, the CPU responds to the interrupt by sending an interrupt acknowledge signal ($\overline{\text{INTA}}$) back to the 8259A. The 8259A responds to $\overline{\text{INTA}}$ by pulling INTR back low, and then putting an 8-bit interrupt type code *nn* onto the data lines.
5. The CPU reads the type code *nn* and automatically executes an int *nn* instruction.
6. The int *nn* instruction causes the machine state to be saved by pushing cs:ip and the flags register on the stack (more is pushed for protected-mode operation of 386 and later CPUs—see Chapter 5). It also sets IF=0 and then does an indirect jump to the appropriate location specified by the IDT; for example, in real mode, the four-byte address at absolute location 4**nn*.

7. The interrupt handler must perform the following actions:
   a) Save *all* registers used.
   b) If multiple devices share the IRQ*n*, determine which device interrupted and clear its interrupt flip-flop.
   c) Reenable interrupts as soon as practical with an sti instruction. Note that an iret also reenables interrupts provided the interrupt flag (IF) pushed by the matching interrupt was 1.
   d) Output an end-of-interrupt (EOI) code to the 8259A.
   e) In edge-triggered configurations (all but PS/2s), if multiple devices share the IRQ*n*, poll the other devices for pending interrupts and service accordingly.
   f) Either end the interrupt handler with an iret to restore cs:ip and the flags, or chain to the previously installed interrupt handler. The second approach is used when the current interrupt handler chooses not to handle the interrupt.
8. The interrupting device must pull the IRQ*n* line back low before it tries to send another interrupt request.

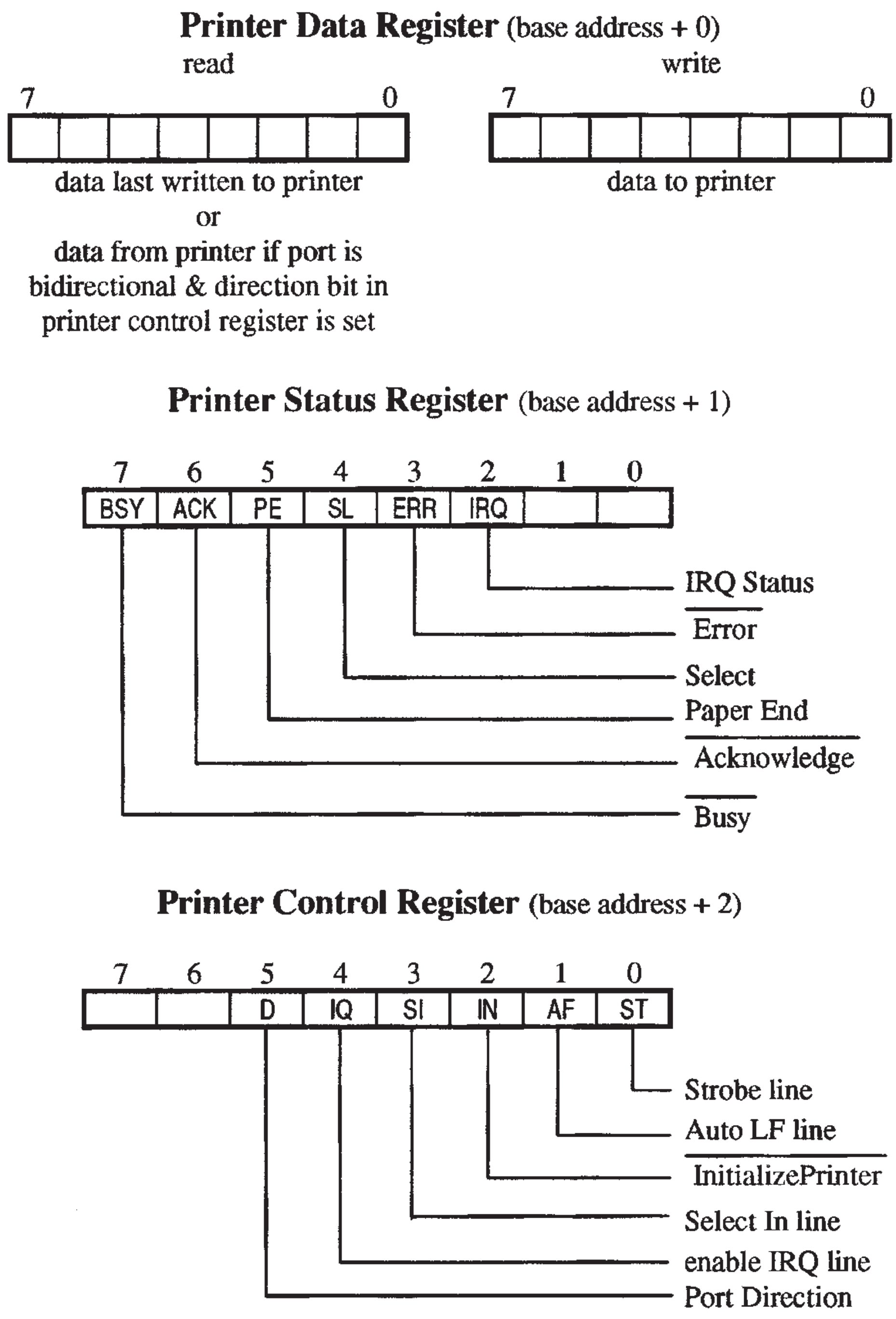
The 8259A has a variety of additional capabilities not generally used in PC operation. For a more complete discussion, see the Intel documentation.

## 7-5. Programmable Interval Timer

A smart peripheral that serves three important purposes on the PC system board is the three-channel 8253/8254 16-bit timer/counter circuit, diagrammed in Figure 7-14. The 8253 was used in the original IBM PC and an equivalent circuit appears in the IBM PS/2s. The somewhat more powerful 8254 was used in the original IBM AT and, except for the PS/2s, is used in virtually all PCs with 286 or later CPUs. Each channel can accept an input clock signal having any frequency from 0 to 2 MHz and produce an output signal whose frequency is the input frequency divided by an arbitrary 16-bit number. On the PC, the input clock frequency for all three channels is 1.1931817 MHz, which is derived from the old system clock, as described shortly.

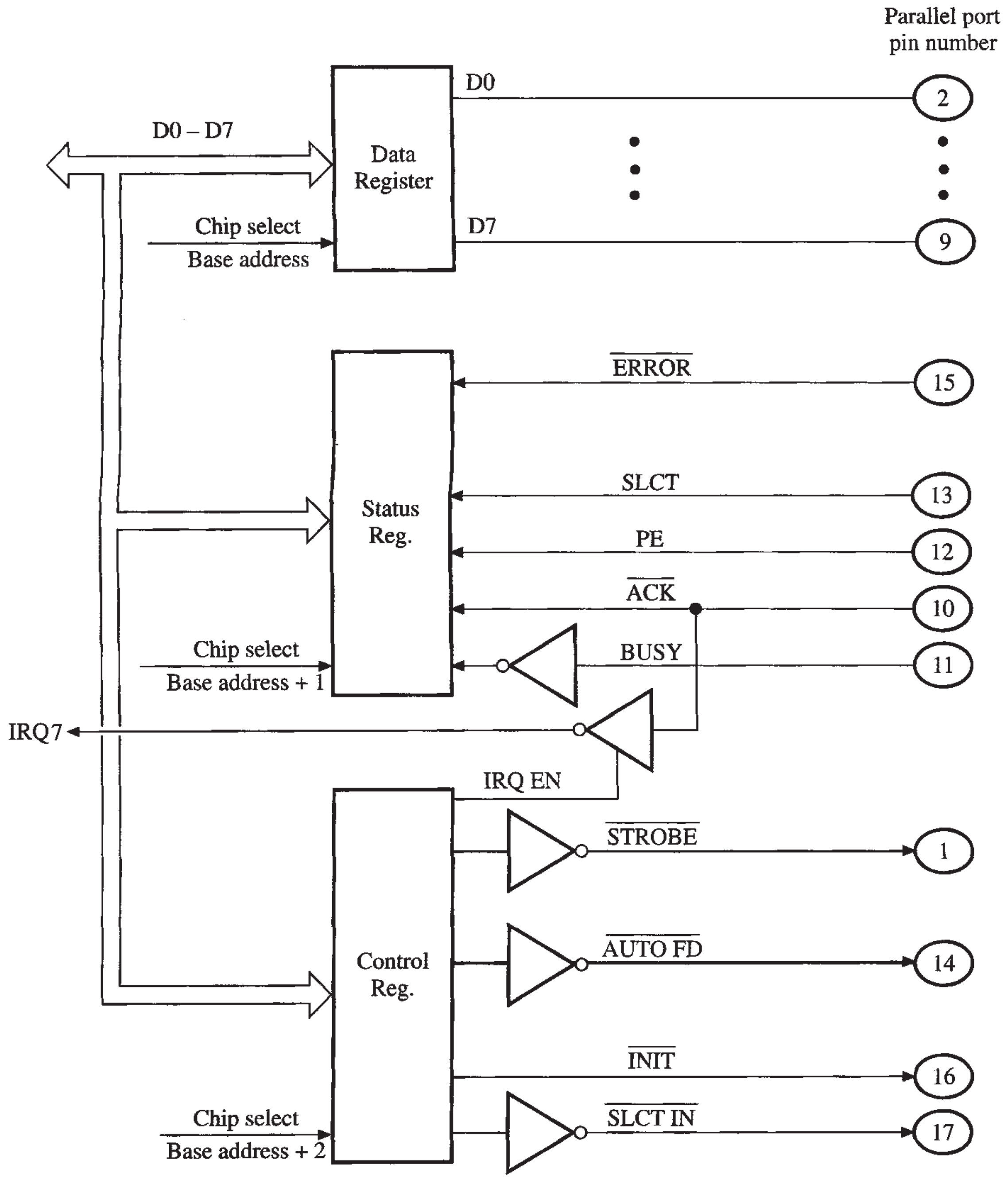
On PCs, the output from channel 0 is used to provide an 18.2-Hz timer interrupt (int 8 by default) that among other things keeps track of the time of day in 18.2-Hz ticks since midnight, a rate given by 1.1931817 MHz divided by 65536. However peculiar the 18.2-Hz tick rate may seem, it's a real standard and it's more or less straightforward to convert it and its higher-speed relatives to more human-oriented time bases as discussed in Section 8-2. The output from channel 1 is used to tell the motherboard refresh circuitry to refresh the dynamic RAMs, and the output from channel 2 is used to send sound to the speaker (see Section 9-1 for discussion and

**FIGURE 11-2.** The three registers that comprise the PC parallel printer port. On read, the printer control register returns the current status of the line bits; on write, it sets the line values as specified by the write byte.


Printer Data Register (base address + 0)
read
write
7
0
7
0
data last written to printer or data from printer if port is bidirectional & direction bit in printer control register is set
data to printer
Printer Status Register (base address + 1)
7 6 5 4 3 2 1 0
BSY ACK PE SL ERR IRQ
IRQ Status
$\overline{\text{Error}}$
Select
Paper End
$\overline{\text{Acknowledge}}$
$\overline{\text{Busy}}$
Printer Control Register (base address + 2)
7 6 5 4 3 2 1 0
D IQ SI IN AF ST
Strobe line
Auto LF line
$\overline{\text{InitializePrinter}}$
Select In line
enable IRQ line
Port Direction

As soon as the BUSY line goes low, the PC ROM BIOS routine pulses the $\overline{\text{STROBE}}$ line low (bit 0 of the control register) telling the printer that a new byte is ready, and then the routine returns. On the standard IBM version of the printer

**FIGURE 11-3.** IBM PC Centronics parallel-port architecture, signals, and output connector pin numbers.



A common but rather unorthodox use of the parallel port is for software copy protection by means of a *dongle*, which is a small box that plugs into the parallel port connector. The copy-protected software checks to see if the dongle is there and won't run if it isn't. At the same time, the dongle doesn't (or at least isn't supposed to!) interfere with the normal use of the parallel port for other devices such as

**TABLE 11-2.** 8250A/16450/16550AF UART register bit definitions. FIFO fields pertain to 16550AF *only. The scratch pad register is missing on the original 8250 UART.* † DLAB (divisor latch address bit) = 0. If DLAB = 1, registers 0 and 1 are the low and high bytes, respectively, of the baud-rate divisor latch. On the dual 16552 UART, register 2 also has different meanings for DLAB = 1 (see van Gilluwe, 1994). Registers are read/write unless otherwise noted in the leftmost column.

| Register | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Receiver Buffer 0 (R)† | data bit 7 | data bit 6 | data bit 5 | data bit 4 | data bit 3 | data bit 2 | data bit 1 | data bit 0 |
| Transmit Buffer 0 (W)† | data bit 7 | data bit 6 | data bit 5 | data bit 4 | data bit 3 | data bit 2 | data bit 1 | data bit 0 |
| Interrupt Enable 1† | 0 | 0 | 0 | 0 | MS int enable | RLS int enable | THRE int enable | RDA int enable |
| Interrupt ID 2 (R) | FIFOs enabled | FIFOs enabled | 0 | 0 | int ID bit 2 | int ID bit 1 | int ID bit 0 | 0 if int pending |
| FIFO Control 2 (W) | receiver trigger bit 1 | receiver trigger bit 0 | reserved | reserved | DMA mode select | xmit FIFO reset | rcvr FIFO reset | FIFO enable |
| Line Control 3 | DLAB | set break | stick parity | even parity | parity enable | # stop bits | length select bit 1 | length select bit 0 |
| Modem Control 4 | 0 | 0 | 0 | loop | OUT2 | OUT1 | RTS | DTR |
| Line Status 5 | receiver FIFO error | xmit empty | THRE | break int | framing error | parity error | overrun error | data ready |
| Modem Status 6 | DCD | RI | DSR | CTS | ΔDCD | TERI | ΔDSR | ΔCTS |
| Scratch pad 7 | data bit 7 | data bit 6 | data bit 5 | data bit 4 | data bit 3 | data bit 2 | data bit 1 | data bit 0 |

The base addresses of the PC serial ports are 3f8h for the first serial port (COM1) and 2f8h for the second serial port (COM2). 3e8h and 2e8h are also used on some systems for COM3 and COM4. The BIOS checks to see which parallel ports are present during its power-on tests and stores the base addresses of the serial ports it finds in the first four words of the BIOS RAM data area (address 40:0h). To access the UART registers in a serial port, the register offset address is added to the port base address. For example, to read the UART line status register for COM1, just execute an in al,dx from port 378h + 5 = 37dh.

You can receive and transmit serial data by simply executing in and out instructions to the receive and transmit data registers, respectively. But before the UART can work, its control and baud-rate divisor registers have to be initialized. For standard settings on the PC, this can be accomplished easily by a call to int 14h call with ah = 0 and with the bits in al set as follows.

| | | |
|---|---|---|
| bits 1,0 | = | word length (10 gives 7 bits, 11 gives 8 bits) |
| bit 2 | = | number of stop bits (0 gives 1 SB, 1 gives 2 SB) |
| bit 3 | = | parity enabled (1) or disabled (0) |
| bit 4 | = | even (1) or odd (0) parity if bit 3 = 1 |
| bits 7,6,5 | = | the values 0 through 111b give baud rates of 110, 150, 300, 600, 1200, 2400, 4800, and 9600 baud respectively |

A popular choice is al = 0e3h, for 9600 baud, no parity, 1 stop bit, and 8-bit words. To set this up you can use

```
mov     ax,0e3h        ;ah = 0 for serial port initialization call
int     14h            ;Use the BIOS routine
```

The alternative is to program the baud-rate divisor latches and the line control register directly as illustrated in the communications program of Section 11-5. In general, you must do this to obtain higher baud rates or to activate the FIFOs on 16550 UARTs. See, for example, Van Gilluwe (1994) for a detailed description of the individual bits in each UART register and extended BIOS functions available on some systems such as the IBM PS/2.

Executing int 14h in the ROM BIOS also allows you to send and receive serial data and to check the serial-port status. Unfortunately, you may not be able to use these routines for anything but modem communications. The problem is that the routines check the handshake lines with a countdown loop that times out too fast for most purposes. This causes characters to be lost if you're sending data to a device that must stop receiving data for anything more than a very brief time. The serial communications functions available with the MS-DOS mode command are more flexible.

To illustrate how you can write your own serial-port drivers, we present some basic, simple working routines (see also Section 4-6). Polling techniques are used rather than interrupts here (see Section 11-5 for more advanced routines). The programs assume that the line-control parameters (stop bits, parity, and word length) and the baud rate have already been set by the int-14h initialization call. To properly send or receive data from the UART's data port (port 3f8h for COM1), you need to know how to control the $\overline{\text{DTR}}$ and $\overline{\text{RTS}}$ output handshake lines, how to monitor the $\overline{\text{DSR}}$

and $\overline{\text{CTS}}$ input handshake lines, and how to detect when serial data has been received or more data can be transmitted.

Control of the output handshake lines is accomplished through the *modem control register* (MCR) located at port 3fch for COM1. To drive the $\overline{\text{DTR}}$ output low, thereby indicating that the UART is operational, you set bit 0 of this register to 1. Similarly, setting bit 1 = 1 forces $\overline{\text{RTS}}$ low, indicating that the UART is ready to receive data. You can pull both lines low with the code

```
mov     dx,3fch         ;Point at COM1 modem control register
in      al,dx           ;Get current state of signals
or      al,3            ;Force DTR and RTS low
jmp     $+2             ;Slow down I/O
out     dx,al
```

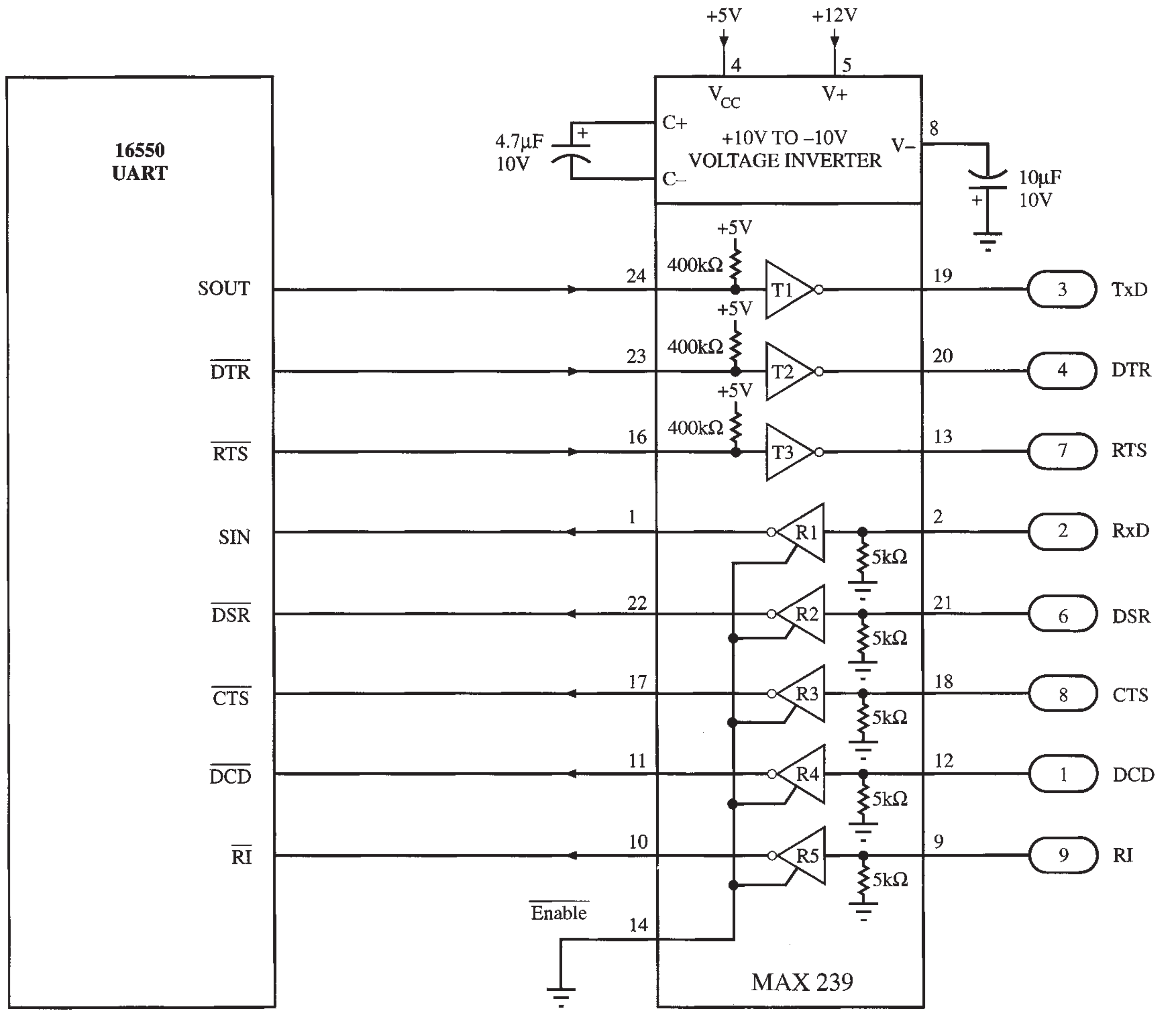
Section 11-5 discusses how to use the $\overline{\text{DTR}}$ line for handshaking; for now, we just make sure both $\overline{\text{DTR}}$ and $\overline{\text{RTS}}$ are low, since the remote device may not send or receive any data unless they are! Similarly, the $\overline{\text{OUT1}}$ and $\overline{\text{OUT2}}$ are forced low by setting bits 2 and 3 high in the modem control register. $\overline{\text{OUT2}}$ must be low for UART interrupts to occur with the PC serial adapters. Setting MCR bit 4 high causes the UART's serial and handshake outputs to be looped back into the corresponding inputs for testing purposes. Bits 6 and 7 are always 0.

The state of the handshake input lines can be read as bits in the *modem status register* (MSR) located at port 3feh for COM1. As for the outputs, everything is inverted so that 1s correspond to low pin values and 0s to high values. Specifically, bits 4–7 give the state of $\overline{\text{CTS}}$, $\overline{\text{DSR}}$, $\overline{\text{RI}}$, and $\overline{\text{DCD}}$, respectively. Bits 0–3 give the corresponding delta or change signals. For example, if the $\overline{\text{CTS}}$ value has changed since the last time the modem status register was read, bit 0 is set to 1. Such changes can also be programmed to cause an interrupt. Section 11-5 shows how to use the $\overline{\text{DSR}}$ input for handshaking. For now, just ignore this register.

The *line status register* (LSR) located at port 3fdh for COM1 is used to monitor the status of the serial lines. The DR bit, bit 0, tells whether data has been received, and the THRE bit, bit 5, tells whether the transmitter holding register is empty. If THRE = 1, the last character sent has been transmitted and you can send another character; if DR = 1, a character has been received and is ready to be read by the CPU. Other bits in the line status register are set to 1 if overrun (bit 1), parity (bit 2), or framing (bit 3) errors have occurred. Bit 4 = 1 announces that a break interrupt has occurred; that is, a space value for longer than a full-word transmission time has been received. Bit 6 = 1 says the transmitter shift register is empty. Having both THRE = 1 and the transmitter shift register empty implies that *two* characters can be immediately sent to the UART. This works because the UART transmitter is *double buffered*, as shown in the block diagram of Figure 11-7. While one character is being shifted out bit-by-bit in the transmitter shift register, a second character can be stored in the transmitter holding register, waiting its turn to be transferred into the shift register as soon as the current character has been shifted out.

also combine drivers and receivers in a single package so that only a single chip needs to be connected to a UART to implement an RS-232 serial port as shown in Figure 11-9. Note that the RS-232 drivers and receivers both are inverters so that the signals on a serial cable are inverted from their values at a UART. Thus, the handshake lines are all active high when observed on the cable. Note also that this interface works even if the cable doesn't connect all the handshake signals since input lines are pulled high by the receiver circuitry if their inputs are disconnected. Many interfaces work this way, requiring you only to connect pins 2, 3, and 5 (RxD, TxD, and signal ground) on a DB-9 connector to make a working serial interface.

**FIGURE 11-9.** Typical RS-232 interface for a PC supporting all the commonly used serial port signals. Note that all signals coming from or going into the UART are inverted.


+5V
+12V
4
5
16550 UART
4.7µF 10V
V<sub>CC</sub>
V+
C+
C–
+10V TO –10V VOLTAGE INVERTER
V–
8
10µF 10V
+5V
400kΩ
SOUT
24
T1
19
3
TxD
DTR
23
T2
20
4
DTR
RTS
16
T3
13
7
RTS
SIN
1
R1
2
2
RxD
5kΩ
DSR
22
R2
21
6
DSR
CTS
17
R3
18
8
CTS
DCD
11
R4
12
1
DCD
RI
10
R5
9
9
RI
Enable
14
MAX 239