THE ART OF ELECTRONICS

Second Edition

Paul Horowitz HARVARD UNIVERSITY

Winfield Hill ROWLAND INSTITUTE FOR SCIENCE, CAMBRIDGE, MASSACHUSETTS

CAMBRIDGE
UNIVERSITY PRESS

MICROCOMPUTERS

CHAPTER 10

MINICOMPUTERS, MICROCOMPUTERS, AND MICROPROCESSORS

The availability of inexpensive ($1k) small computers has made it attractive to control experiments and processes, collect data, and perform computation directly under the control of a computer. Small computers are commonly used in laboratory and industrial settings, and knowledge of their capabilities, program languages, and interfacing requirements is an essential part of electronics know-how.

The microcomputer evolved from the earlier *minicomputer*, a small machine whose central processing unit (CPU) was constructed from SSI and MSI ICs, usually occupying one or more large printed-circuit boards. As large-scale integration improved, it became possible to put minicomputer CPU performance into a single LSI chip; thus a *microcomputer* is a computer whose CPU is constructed from just a few (often only one) LSI microcircuits; the CPU chip (or chip set) constitutes a *microprocessor*. For example, DEC's popular PDP-11 minicomputers (CPU on several interconnected boards) were succeeded by a family of similarly named computers whose CPUs were built from a few LSI chips in place of many SSI/MSI chips; at about the same time, Motorola introduced a high-performance microprocessor (the 68000) that has many similarities to the PDP-11 and was obviously influenced by it.

Most modern small computers are in fact microcomputers, relying on the impressive performance of the present generation of microprocessors. The phrase "superminicomputer" has recently surfaced and seems to signify a class of machines that achieve higher performance, in some cases rivaling the large and expensive "mainframe" computers. In some cases the distinction refers more to physical size or number of peripherals than to the scale of integration used in the construction of the CPU.

A more important distinction separates microcomputers from *microcontrollers*, a term used to describe the use of a microprocessor, along with a small amount of memory and other support chips, for dedicated control of a process or instrument. In this role a microprocessor plus a few assorted chips and some ROM (read-only memory) can flexibly replace a complicated logic circuit of gates, flip-flops, and analog/digital conversion functions and should be considered whenever embarking on a large design project. There are microprocessors optimized for this kind of application, generally characterized by on-chip timers, ports, and other functions that usually require extra ICs, at the expense of the computational power and large address space that characterizes microprocessors intended for microcomputer-based computational tasks.

In this chapter we will describe microcomputer architecture, programming, and interfacing, with some examples of useful and simple interfacing of peripherals to the IBM PC/XT (here we refer to the original PC bus and its derivatives such as the PC/AT and compatibles, and the low end of the PS/2 line). Most of the ideas introduced in this chapter will carry over to the next chapter, where we will get into a detailed discussion of the selection and construction of microprocessor-based circuits and systems; for those examples we will use the 68008 microprocessor, a member of the Motorola 68000 family that, together with the Intel 8086 family, dominates small computers. Generally speaking, with microcomputers the design of the computer itself, including the integration of memory, disks, and I/O control, as well as system programming and utility program development, is taken care of by the manufacturer (and suppliers of compatible hardware and software). The user need only worry about special-purpose interfaces and the job of user programming. By contrast, in a dedicated microprocessor system, the choices of memory types, system interconnection, and programming generally have to be made by the designer. Microcomputer manufacturers are generally committed to providing system and utility software as part of a complete computing system (often including peripherals), whereas the microprocessor manufacturers (semiconductor companies) generally see the design and marketing of microprocessor and support chips as their central tasks. In this chapter, then, we will describe computer architecture and programming and will concentrate on the details of internal communication and interfacing.

10.01 Computer architecture

Figure 10.1 summarizes the organization typical of most computers. Let's take it from left to right:

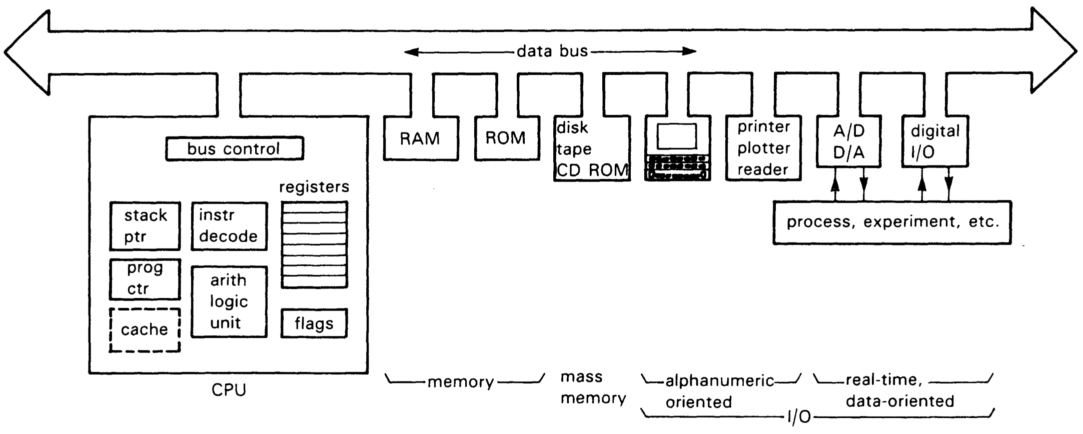


Figure 10.1. Block diagram of a computer.

CPU

The central processing unit, or CPU, is the heart of the machine. Computers do their computation in the CPU on chunks of data organized as computer *words*. Word size can range from 4 bits to 32 bits or more, with a 16-bit word size being the most popular in current microcomputers. A *byte* is 8 bits (half a byte, or 4 bits, is sometimes called a "nybble"). A portion of the CPU called the *instruction decoder* interprets the successive instructions (fetched from memory), figuring out what should be done in each case. The CPU has an *arithmetic unit*, which can perform the instructed operations, such as add, complement, compare, shift, move, etc., on quantities contained in *registers* (and sometimes in *memory*). The *program counter* keeps track of the current location in the executing program. It normally increments after each instruction, but it can take on a new value after a "jump" or "branch" instruction. The *bus control circuitry* handles communication with memory and I/O. Most computers also have a *stack pointer register* (more on that later) and a few *flags* (carry, zero, sign) that get tested for conditional branching. Many high-performance processors also include *cache* memory, which holds values recently fetched from memory for quicker access.

There has been a lot of activity in the experimental field of "parallel processing," in which you interconnect many CPUs to get tremendous computational power. With time this trend may become dominant in high-performance processing. For the time being, however, our single-CPU machine, executing instructions serially, represents the standard microcomputer architecture.

Memory

All computers have some fast random-access memory, called RAM (it used to be called "core," because tiny magnetic cores held the data, one bit per core). In a large microcomputer this may include 10 megabytes or more, although a megabyte is more typical, and as little as 16K may be used in a microcontroller. (When used to describe memory sizes, K doesn't mean 1000, but rather 1024, or 2^{10}; thus, 16K bytes is actually 16,384 bytes. We employ the lower-case symbol k to mean 1000.) This memory can typically be read and written in about 100ns. RAM is almost always *volatile*, which means that its information evaporates when power is removed (maybe it should be called "forgettory"!). All computers therefore include some nonvolatile memory, usually ROM (read-only memory), to "bootstrap" the computer, i.e., get it started from a state of total amnesia when power is first turned on. Additional ROM is often programmed with system routines, graphics routines, and other programs that you want to be there all the time.

To get or store information in memory, the CPU "addresses" the desired word. Most computers address memory by bytes, beginning at byte 0 and going sequentially through to the last byte in memory. Since most computer words are several bytes long, you are usually storing or fetching a group of bytes at a time; this is usually expedited by having a data bus that is several bytes wide. For example, microcomputers that use the 80386 or 68020 use a bus 32 bits (4 bytes) wide, so that a 32-bit word can be moved to/from memory in one memory fetch. (There are control signals to specify how many contiguous bytes are being moved, since even with a large bus you may want only 1 or 2 bytes.)

In a computer with lots of memory, it takes three or four bytes to specify an arbitrary memory address anywhere in the machine. Since most memory references in an actual program are usually "nearby," all computers provide for simplified addressing modes: "Relative" addressing specifies an address by its distance from the present instruction; "indirect" addressing uses the contents of a CPU register to point to a location in memory; "paged" addressing uses a shortened address to refer to a memory location within a small area (a page); "direct" or "absolute" addressing uses the next few bytes in memory to specify an address. A modern CPU embellishes this short list with additional "indexed," "autoincrementing," and other useful addressing modes, which we'll learn about in the next chapter.

Both programs and data are kept in memory during program execution. The CPU fetches instructions from memory, figures out what they mean, and does the appropriate things, often involving data stored somewhere else in memory. General-purpose computers usually store programs and data in the same memory, and in fact the computer doesn't even know one from the other. Amusing things start to happen if a program goes awry and you "execute" data!

Since computer programs spend most of their time looping through a relatively short sequence of instructions, you can enhance performance by providing a small, but fast, *cache* memory, in which you routinely store copies of the most recently used memory locations. A cached CPU checks its local cache first, before fetching from (slower) main memory; when looping through familiar territory, you often achieve a cache "hit" rate of 95% or better, dramatically improving execution speed.

Mass memory

Computers intended for program development or computation, as opposed to dedicated control processors, usually have one or more mass-storage devices. "Hard" disks (also called "Winchester") and "floppy" disks ("diskettes") are the usual ones, with storage capacities going from a few hundred kilobytes to a few megabytes (floppy disks), and from a few tens of megabytes to a few hundred megabytes (hard disks). Most well-endowed computers also have a tape drive or two, ranging from a simple cartridge-tape "streaming" drive to a full-fledged 9-track, half-inch, large-reel tape (the kind that are always spinning in the background in science fiction movies). A newer technology uses 8mm videotape (the kind that lives in those little handheld video cameras) to store a gigabyte on a small tape cartridge. And the latest in mass storage is the "CD ROM," which uses the same optical disk technology as audio CDs (compact discs); they store 600 megabytes on one side of a 5 inch plastic disk, with much faster access than any tape medium. Unlike audio CDs, there are CD ROM drives that let you write as well as read, by laser-burning pits in a blank CD; they're called "WORM," for "write once, read many." Furthermore, fully erasable read/write magneto-optic disk memories are also available.

Compared with RAM, mass-storage media are generally slow, magnetic tape being the slowest, with access times of many seconds, and hard disks being the fastest (and most expensive), with average access times of tens of milliseconds. With all mass-storage devices, data transfer is rapid (10K to 100K bytes per second or more) once the data has been located. You generally keep programs, data files, plot files, etc., on some sort of mass-storage device and bring these into RAM only when doing computation. Many users can simultaneously fit their programs on one disk; a moderate-size optical disk can hold the contents of the *Encyclopaedia Britannica* several times over.

If your computer has lots of RAM, a nice way to speed up computer operations that make heavy use of disk is to form a "RAM disk" by loading all the relevant disk files into RAM when you start. Thus you might put a text editor, compiler, and linker/loader into RAM; then you can switch back and forth without waiting for the disk. Be careful, though; because none of your work is being saved on nonvolatile disk, you lose all your work if the computer crashes.

Alphanumeric and graphic I/O

It is nice to have a powerful computer, capable of millions of smart computations per second, but it doesn't do you any good if it keeps all its results to itself. Peripherals such as a keyboard and screen (the combination is a "terminal"), "mouse," printer, etc., let man and machine communicate, and these are essential in any "friendly" computer system. These peripherals are mostly oriented toward programming, word processing, spreadsheets, and graphics; you use them when writing programs, debugging, listing, writing and printing documents, manipulating quantities and objects, and playing flight simulator. These sorts of peripherals, together with suitable interfaces, are available from many sources, including the microcomputer manufacturer.

Real-time I/O

For experiment or process control and data logging, or for exotic applications such as speech or music synthesis, you need A/D and D/A devices that can communicate with the computer in "real time," i.e., while things are happening. The possibilities are almost endless here, although a general-purpose set of multiplexed A/D converters, a few fast D/As, and some digital "ports" (serial or parallel) for exchange of digital data will permit many interesting applications. Such general-purpose peripherals are commercially available for most popular computer buses. If you want something fancier, such as improved performance (higher speed, more channels) or special-purpose functions (tone generation, frequency synthesis, time-interval generation, etc.), you may have to build it yourself. This is where a knowledge of bus interfacing and programming techniques is essential, though it's helpful in any case.

Network interface

Powerful desktop computers become even more powerful when they can exchange files with other computers. One way to do this is to "log on" to a remote computer via telephone lines, then use the features of the remote computer that you need. That might include access to a large data base or special programs, a powerful supercomputer, computer "mail," or a colleague's text or data file. For these purposes you need a "modem" (modulator/demodulator), which either plugs directly into your computer's bus or hooks onto a serial data port. We'll have more to say about this later.

Another way to extend the scope of your machine is to use a local area network (LAN) to link a group of computers together. An example is Ethernet, which provides communication at rates up to 10Mb/s among linked machines, via a single coaxial cable. A LAN lets you access files on anyone's machine; in fact, with a good LAN you would probably pool your resources, sharing a fast large disk, high-priced plotters and printers, etc. Each "workstation" would then have only limited mass storage, but enough computational and display capability for the work you want to do with it. Such a setup is ideal for a publishing house or newspaper, for example, where different people work on manuscripts as they are readied for publication. You can get Ethernet (and other LAN) interfaces for most microcomputers.

Data bus

For communication within the computer between the CPU and memory or peripherals, all computers use a *bus*, a set of shared lines for exchange of digital words. (Many buses also allow communication *between* peripherals, though this capability is used less often.) The use of a shared bus vastly simplifies interconnections, since otherwise you would need multiwire cables connecting every pair of communicating devices. With a little care in bus design and implementation, everything works fine.

The bus contains a set of DATA lines (generally the same number as bits in a word - 8 for microcontrollers and low-performance PCs, 16 or 32 for more sophisticated microcomputers), some ADDRESS lines for determining who should "talk" or "listen" on the line, and a bunch of CONTROL lines that specify what action is going on [data going to or from the CPU, interrupt handling, DMA (direct memory access) transfers, etc.]. All the DATA lines, as well as a number of others, are *bidirectional* - they're driven by three-state devices, or in some cases by open-collector gates with resistor pullups somewhere (usually at the end of the bus, where they also serve as terminators to minimize reflections, see Section 13.09); pullups may be necessary with three-state drivers also, if the bus is physically long.

Three-state or open-collector devices are used so that devices connected to the bus can disable their bus drivers, since in normal operation only one device is asserting data onto the bus at any time. Each computer has a well-defined protocol for determining who asserts data, and when. If it didn't, total chaos would result, with everyone shouting at once (so to speak). (Computer people can't resist personalizing their machines, peripherals, etc. Engineers are even worse, with flip-flops and even gates coming to life. Naturally, we follow the trend.)

There is one interesting distinction in computer buses. They can be either *synchronous* or *asynchronous*, with examples of each in currently popular microcomputers. You will see what this means when we get into the details of communication via the bus.

We'll return to the bus in detail, with interface examples, using the example of the popular IBM PC/XT family. First, though, we need to look at the CPU's instruction set.

A COMPUTER INSTRUCTION SET

10.02 Assembly language and machine language

In order to understand bus signals and computer interfacing, you've got to understand what the CPU does when it executes various instructions. At this point, therefore, we would like to introduce the instruction set that goes with the IBM PC/XT family. Unfortunately, the instruction sets of most real-world microprocessors tend to be rich with complexities and extra features, and the Intel 8086 series is no exception. However, since our purpose is only to illustrate bus signals and interfacing (not fancy programming), we'll take a shortcut by laying out a subset of 8086 instructions. By leaving out the "extra" instructions we'll wind up with a compact set of instructions that is both understandable and complete enough to do any programming task. We'll then use it to show some examples of interfacing and programming. These examples will help convey the idea of programming at the "machine-language" level, something quite different from programming in a high-level language like FORTRAN or C.

First, a word on "machine language" and "assembly language." As we mentioned earlier, the computer's CPU is designed to interpret certain words as instructions and carry out the appointed tasks. This "machine language" consists of a set of binary instructions, each of which may occupy one or more bytes. Incrementing (increasing by one) the contents of a CPU register would be a single-byte instruction, for example, whereas loading a register with the contents of a memory location would usually require at least two bytes, perhaps as many as five (the first would specify the operation and register destination, and four more would be necessary to specify an arbitrary memory location in a large machine). It is a sad fact of life that different computers have different machine languages, and there is no standard whatsoever.

Programming directly in machine language is extremely tedious, since you wind up dealing with columns of binary numbers, each bit of which has to be bit-perfect, so to speak. For this reason you invariably use a program called an *assembler*; it allows you to write programs using easily remembered mnemonics for the instructions, and symbolic names of your own choosing for memory locations and variables. This *assembly-language* program, really nothing more than a number of cryptic-looking lines of letters and numbers, is massaged by a program called an *assembler* to produce as its output a finished program in machine-language *object code* that the computer can execute. Each line of assembly code gets turned into a few machine-language bytes (1 to 6 bytes, for the 8086). The computer cannot execute assembly-language instructions directly. To make these ideas concrete, let's look at our subset of the 8086/8 assembly language and do a few examples.

10.03 Simplified 8086/8 instruction set

The 8086 is a 16-bit processor with a rich, and somewhat idiosyncratic, instruction set; part of its complexity stems from the designers' objective to maintain compatibility with the earlier 8080 8-bit processor. Newer CPUs, such as the 80286 and 80386, can still execute the full 8086 instruction set. We've gone through the instructions with a machete, keeping 10 arithmetic operations and 11 others. Here they are:

Instruction	What you call it	What it does
arithmetic		
`MOV` *b,a*	move	*a*→*b*; *a* unchanged
`ADD` *b,a*	add	*a*+*b*→*b*; *a* unchanged
`SUB` *b,a*	subtract	*b*–*a*→*b*; *a* unchanged
`AND` *b,a*	and	*a* AND *b*→*b* bitwise; *a* unchanged
`OR` *b,a*	or	*a* OR *b*→*b* bitwise; *a* unchanged
`CMP` *b,a*	compare	set flags as if *b*–*a*; *a,b* unchanged
`INC` *rm*	increment	*rm*+1→*rm*
`DEC` *rm*	decrement	*rm*–1→*rm*
`NOT` *rm*	not	1's complement of *rm*→*rm*
`NEG` *rm*	negate	negative (2's comp) of *rm*→*rm*
stack		
`PUSH` *rm*	push	push *rm* onto stack (2 bytes)
`POP` *rm*	pop	pop 2 bytes from stack to *rm*
control		
`JMP` *label*	jump	jump to instr *label*
`Jcc` *label*	jump conditional	jump to instr *label* if *cc* true
`CALL` *label*	call	push next adr, jump to instr *label*
`RET`	return	pop stack, jump to that adr
`IRET`	return from int	pop stack, restore flags, return
`STI`	set interrupt	enable interrupts
`CLI`	clear interrupt	disable interrupts
input/output		
`IN AX(AL)`,*port*	input	*port*→AX (or AL)
`OUT` *port*,`AX(AL)`	output	AX (or AL)→*port*

notes

b,a: any of *m,r r,m r,r m,imm r,imm*
rm: *r* or *m*, via various addressing modes
cc: any of `Z NZ G GE LE L C NC`
label: via various addressing modes
port: byte (via *imm*) or word (via `DX`)

A quick tour

Some explanations: The first six arithmetic instructions operate on pairs of numbers ("2-operand" instructions), which we've abbreviated as *b,a*, and which can be any of the 5 pairs listed in the notes; *m* means the contents of a memory location, *r* means the contents of a CPU register (there are 8), and *imm* means an *immediate* argument, which is a number stored in the next 1 to 4 bytes of memory following the

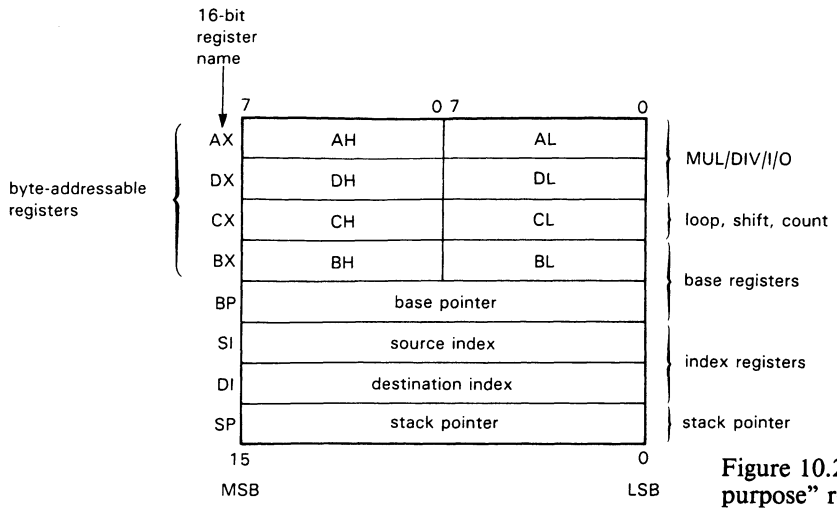


Figure 10.2. 8086 "general-purpose" registers.

instruction. Thus, for example, the instructions

```
MOV   count,CX
ADD   small,02H
AND   AX,007FH
```

have arguments of the form *m,r*, *m,imm*, and *r,imm*, respectively. The first copies the contents of register CX to a memory location that we've named "count"; the second adds 2 to the contents of another memory location called "small"; the third clears the top 9 bits of 16-bit register AX, while preserving the bottom 7 bits unchanged (a so-called masking operation). Note Intel's argument convention: The first argument is replaced or modified by the second argument. (In the next chapter we'll learn that Motorola decided to do it the other way around!)

The last four arithmetic operations take only a single operand, which can be either the contents of a register or memory. Here are two examples:

```
INC   count
NEG   AL
```

The first adds 1 to the contents of memory location "count," while the second changes the sign of register AL.

A detour: addressing

Before continuing, a word on registers and memory addressing. The 8086 claims to have 8 "general-purpose" registers, but after reading the fine print you'll realize that most of them have special uses (Fig. 10.2). Four of them (A–D) can be used either as single 16-bit registers (e.g., AX; think of "X" as "extended") or as a pair of byte registers (AH, AL; "high" and "low" halves). The BX and BP registers can hold addresses, as can the SI and DI registers, and tend to be used for addressing (see below). Special looping instructions (which we omitted from our short list) use register C, while multiply/divide and I/O instructions make analogous use of registers A and D.

Data used in instructions can be an immediate constant, a value held in a register, or a value in memory. You specify immediates by value, and registers by name, as in the examples above. To address memory, the 8086 provides six addressing modes, three of which are described by the diagrams in Figure 10.3. You can just name the variable *directly*, in which case its address gets assembled as a pair of bytes immediately following the instruction; you can put the variable's address in an addressing register (BX, BP, SI, or DI), then use an instruction that specifies addressing *indirectly* through the register; or you can combine the above, adding an immediate *displacement* to the value in a designated addressing register to get the variable's address. The indirect

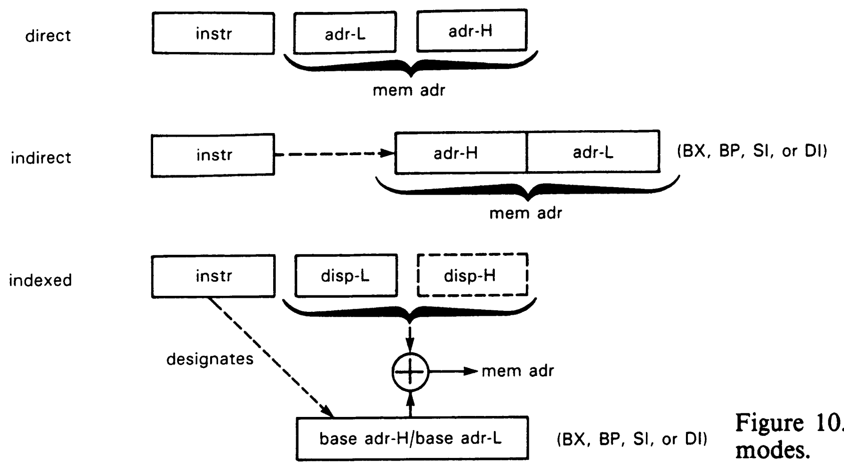


Figure 10.3. Some addressing modes.

mode is faster (assuming the address has already been loaded into an addressing register) and much better if you want to do something to a whole set of numbers (a *string* or *array*). Here are a few addressing examples:

```
MOV  count,100H      (direct,immediate)
MOV  [BX],100H       (indirect,immediate)
MOV  [BX+1000H],AX   (indexed,register)
```

The last two assume you've already put an address into BX. The last instruction copies the contents of AX to a memory location 4K (1000 hex) higher than BX points in memory; we'll give an example shortly showing how you could use this to copy an array.

There's one other complexity of 8086 memory addressing that we've swept under the rug: The "address" generated by any of the above addressing modes is not actually the final address, as should be obvious from the fact that it has only 16 bits (which can address only 64K bytes of memory). In fact, it's called an *offset*; to get an actual address, you add to the offset a 20-bit *base* formed by shifting left 4 bits the contents of a 16-bit *segment register* (there are four such registers). In other words, the 8086 lets you access groups of 64K bytes of memory at a time, with the location of those "segments" within a total memory size of 1Mbyte set by the contents of the segment registers. The use of 16-bit addressing in the 8086 was basically a big mistake, inherited from earlier generations of microprocessors. Newer processors (80386 onward, and the 68000 series) are done right, with 32-bit addressing throughout. Rather than complicate our examples, we'll simply ignore segments entirely; in real life you would, of course, have to worry about them.

Instruction set tour (continued)

The *stack* instructions PUSH and POP come next. A stack is a portion of memory, organized in a special way: When you put data onto the stack (a *push*), it goes into the next available spot ("top" of the stack); and when you retrieve data (a *pop*), it is taken from the top, i.e., it is the item last pushed onto the stack. Thus a stack is a consecutive list of data, stored last-in, first-out (LIFO). It may help to think of a bus driver's coin dispenser (or a lunchroom tray dispenser).

Figure 10.4 shows how it works. The stack lives in ordinary RAM, with the CPU's *stack pointer* (SP) keeping track of the location of the current "top" of the stack. The 8086 stack holds 16-bit words and grows *down* in memory as you push data onto it. The SP is automatically decremented by 2 before each PUSH, and incremented by 2 after each POP. Thus, in

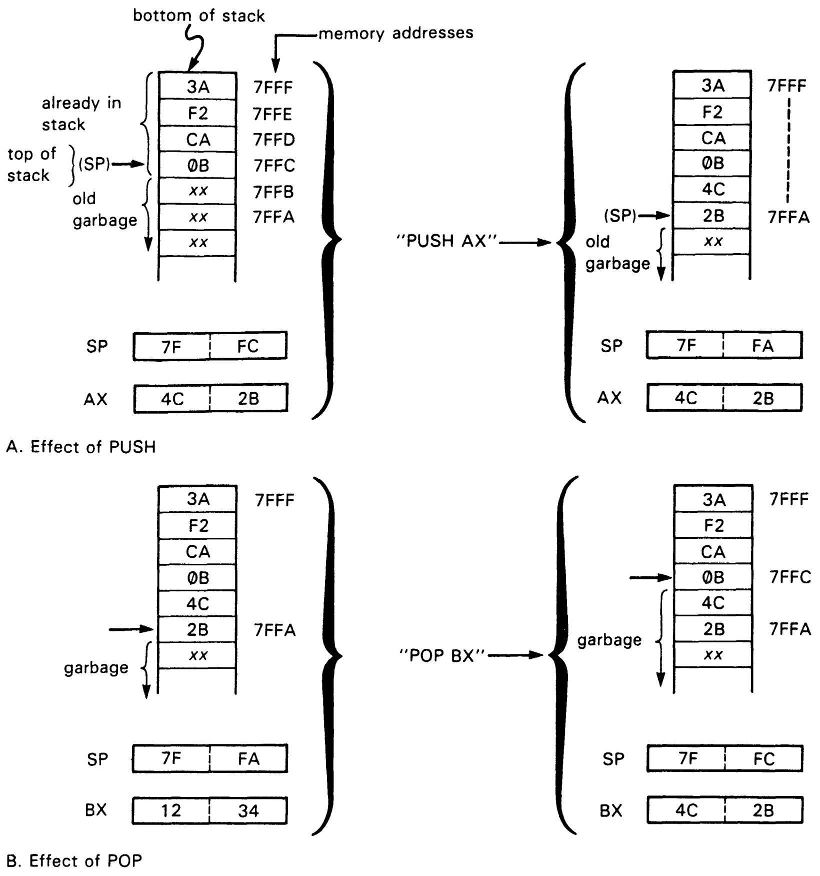


Figure 10.4. Stack operation.

the example, the 16-bit data in register AX is copied onto the top of the stack by the instruction PUSH AX; the SP is left pointing at the last byte pushed. POP reverses the process, as shown. As we will see, the stack plays a central role in subroutine calls and interrupts.

JMP causes the CPU to depart from its usual habit of executing instructions in sequential order, detouring instead to the instruction that you jump to. Conditional jumps (there are eight possibilities, indicated generically as J*cc*) test the flag register (which lives in the CPU, and whose bits are set according to the result of the most recent arithmetic operation), then either jump (if the condition is true) or execute the next instruction in sequence (if the condition is not true). Program 10.1 shows an example. It copies 100 words from the array beginning at 1000 hex to a new array beginning 1K bytes (400H) higher.

Program 10.1

```
       MOV  BX,1000H          ;put array address in BX
       MOV  CL,100            ;initialize loop counter
LOOP:  MOV  AX,[BX]           ;copy array element to AX
       MOV  [BX+400H],AX      ;then to new array
       ADD  BX,2              ;increment array pointer
       DEC  CL                ;decrement counter
       JNZ  LOOP              ;loop if count not zero
NEXT:  (next statement)       ;exit here when done
```

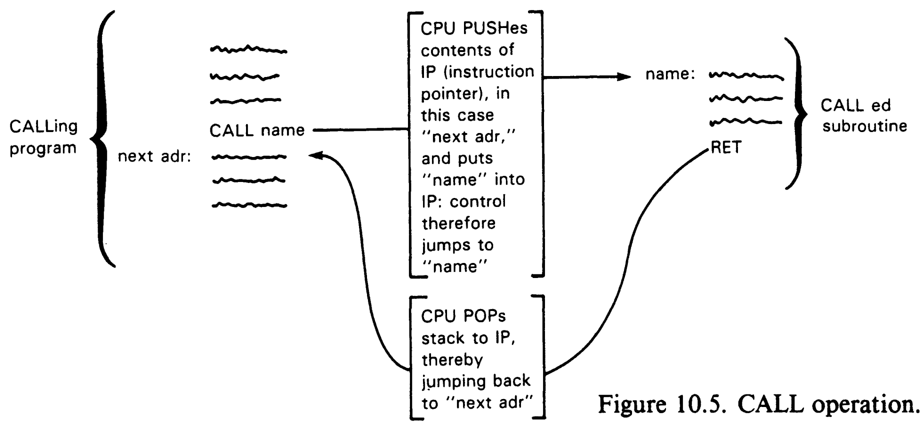

CALLing program
next adr:
CALL name
CPU PUSHes contents of IP (instruction pointer), in this case "next adr," and puts "name" into IP: control therefore jumps to "name"
CPU POPs stack to IP, thereby jumping back to "next adr"
name:
RET
CALL ed subroutine

Figure 10.5. CALL operation.

Note the explicit loading of the pointer (to address register BX) and the loop count (to CL). The actual array of words had to move through a register (we chose AX) because the 8086 does not permit memory-to-memory operations (see the instruction set notes). At the end of the 100th pass through the loop, CZ = 0, and the jump nonzero (JNZ) instruction no longer jumps. This example will work, but in practice you would probably use one of the 8086's faster *string move* instructions. Also, it's good programming practice to use symbolic names for sizes and arrays, rather than constants like 400H and 1000H.

The CALL statement is a subroutine call; it's like a jump, except that the return address (the address of the instruction that would have come next, except for the intervening CALL) is pushed onto the stack. At the end of the subroutine you execute a RET statement, which pops the stack so the program can find its way home (Fig. 10.5). The three statements STI, CLI, and IRET have to do with interrupts, which we'll illustrate with a circuit example later in the chapter. Finally, the I/O instructions IN and OUT move a word or byte between the A register and the addressed port; more on this shortly.

10.04 A programming example

As the example above suggests, assembly language tends to verbosity, with a lot of little steps needed to do a basically simple thing. Here's another example: Suppose you want to increment a number, N, if it equals another number, M. This will typically be a tiny step in a larger program, and in higher level languages it will be a single instruction:

```
if (n==m) ++n;          (C)
IF (N.EQ.M) N=N+1       (Fortran)
if n=m then n:=n+1;     (Pascal), etc.
```

In 8086 assembler, it looks like Program 10.2. The assembler program will

Program 10.2

```
n  DW 0                         ;n (a "word") lives here, and
m  DW 0                         ;m lives here, both initialized to 0

         MOV  AX,n              ;get n
         CMP  AX,m              ;compare
         JNZ  NEXT              ;unequal, do nothing
         INC  n                 ;equal, increment n
NEXT:    (next statement)
           o
           o
           o
```

convert this set of mnemonics to machine language, generally translating each line of assembler *source code* to several machine-language bytes, and the resultant machine-language code will get loaded into successive locations in memory before being executed. Note that it is necessary to tell the assembler to assign some storage space for variables. This you do with the assembler *pseudo-op* "DW" (define word) (*pseudo*-op because it doesn't produce executable code). Unique symbolic labels (e.g., NEXT) can be used to tag instructions; this is usually done only if there is a jump to that location (JNZ NEXT). Giving some locations understandable (to you!) names and adding comments (separated by a semicolon) make the job of programming easier; it also means that you have a chance of understanding what you've written a few weeks later. Programming in assembly language can still be a nuisance, but it is often necessary to write short routines in it, callable from a higher-level language, to handle I/O. Assembly-language programs run faster than programs compiled from a higher language, so it is often used where speed is crucial (e.g., the innermost loop of a long numerical calculation). To some extent the development of the powerful C programming language has minimized the occasions when you must use assembly code. In any case, you can't really understand computer interfacing without understanding the nature of assembly-language I/O. The correspondence between mnemonic assembly language and executable machine language is explored further in Section 11.03, in that case illustrated by 68000 microprocessor programming.

BUS SIGNALS AND INTERFACING

A typical microcomputer data bus has about 50–100 signal lines, devoted to the transfer of data, addresses, and control signals. The IBM PC/XT is typical of a small machine, with 53 signal lines and 8 power/ground lines. Rather than throw them all at you at once, we will approach the subject by building up the bus, beginning with the signal lines necessary for the simplest kind of data interchange (programmed I/O) and adding additional signal lines as they become necessary. We will give some useful interface examples as we go along, to keep things comprehensible and interesting.

10.05 Fundamental bus signals: data, address, strobe

To move data on a shared (party-line) bus, you have to be able to specify the data, the recipient, and the moment when data is valid. Thus, a minimum bus must have DATA lines (for the data to be transferred), ADDRESS lines (to identify the I/O device or memory address), and some STROBE lines (which tell when data is being transferred). There are usually as many DATA lines as bits in the computer word, so a whole word can be transferred at once. In the PC, however, there are only 8 DATA lines (D0–D7); you can move a byte in one transfer, but to move a 16-bit word you have to do two transfers. The number of ADDRESS lines determines the number of addressable devices: If the bus is used for both I/O and memory (the usual situation) there will be 16 to 32 ADDRESS lines (corresponding to a 64Kbyte to 4Gbyte address space); a bus used for I/O only might have 8 to 16 ADDRESS bits (256 to 64K I/O devices). [The IBM PC talks to both memory and I/O on its bus, and has 20 ADDRESS lines (A0–A19), corresponding to a 1Mbyte address space.] Finally, data transfer itself is synchronized by pulses on additional "strobing" bus lines. There are two ways in which this can be done: by having separate READ and WRITE lines, with a pulse on one or the other synchronizing data transfer; or by having one STROBE line and one READ/WRITE′ line, with a pulse on STROBE synchronizing data transfer in a direction specified by the level on the READ/WRITE′ line. The IBM PC uses the first scheme, with (active-LOW) read/write lines called IOR′, IOW′, MEMR′, and MEMW′; there are four because the PC distinguishes between memory and I/O, with individual pairs of read/write strobes for each.

These bus signals - DATA, ADDRESS, and the four strobes - would normally be all you need to do the simplest kind of data transfers. However, on the PC bus you need one more, called ADDRESS ENABLE (AEN), to distinguish normal I/O transfers from what's called "direct memory access" (DMA). We'll get to DMA in Section 10.12; for now, all you need to know is that AEN is LOW for normal I/O, and HIGH for DMA. We now have 33 bus signals: D0–D7, A0–A19, IOR′, IOW′, MEMR′, MEMW′, and AEN. Let's see how they work.

10.06 Programmed I/O: data out

The simplest method of data exchange on a computer bus is known as "programmed I/O," meaning that data is transferred via an `IN` or `OUT` statement in the program (the directions for `IN` and `OUT` are among the few things on which all computer manufacturers agree: `IN` always means *toward* the CPU, and `OUT` always means *from* the CPU). The whole process of data `OUT` (and memory write) is extremely simple and logical (Fig. 10.6). The ADDRESS of the recipient and the DATA to be sent are put onto the respective bus lines by the CPU. A write strobe (IOW′ or MEMW′) is asserted (LOW) by the CPU to signal the recipient that data is good. On the PC's bus the address is guaranteed valid beginning about 100ns before IOW′, and the data are guaranteed valid at least 500ns before the end of IOW′ (and for another 185ns thereafter). To play the game, the peripheral (in this case, an XY "vector" scope display) looks at the ADDRESS and DATA lines. When it sees its own address, it latches the information on the DATA lines, using the trailing edge of the IOW′ pulse as a clocking signal. That's all there is to it.

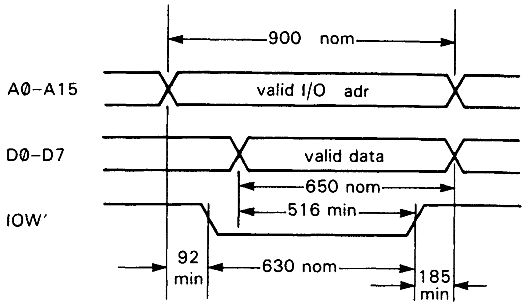


Figure 10.6. I/O WRITE cycle.

Let's look at the example shown in Figure 10.7. Here we have designed an XY scope display; you send it successive X, Y pairs of numbers, and it plots each point in turn on an XY display oscilloscope. First we have to pick an I/O address. Figure 10.8 shows the reserved and available I/O addresses on the IBM PC; we've chosen $3C0_H$ for the X register, and $3C1_H$ for the Y register. The '688 is an octal comparator with enable and LOW-true output on equality, giving a LOW output when the eight high-order bits A2–A9 match the fixed comparison bits, in this case when the address bus contains addresses 3C0–3C3 (you could use a bunch of gates, but an address comparator is more compact). We've also required AEN to be LOW, as explained earlier. The 3-input NANDs complete the address decoding, using A0 and A1, to give LOW outputs on individual addresses 3C0 and 3C1 (another method will be described shortly). Finally, these outputs are ANDed with IOW′ to get the clocks for the X and Y registers, which are '574 octal *D* flip-flops. These latch bytes from the data bus when (a) the correct address is present, (b) AEN is LOW, and (c) an IOW′ is sent. The 8-bit DACs convert the latched bytes to analog voltages, to drive the X and Y inputs

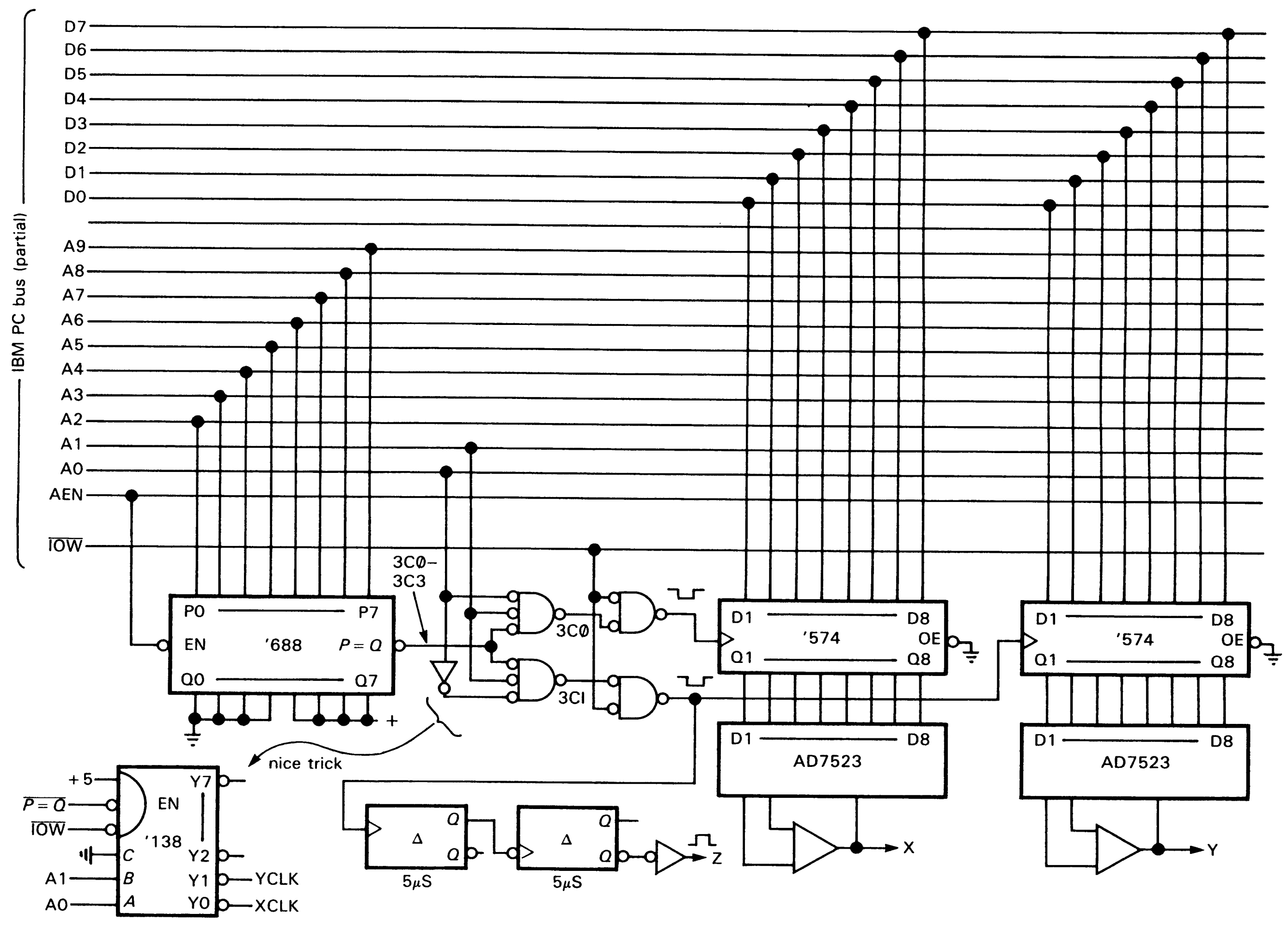


Figure 10.7. XY scope display.

of a display scope. A pair of monostables generates a 5μs "unblanking" pulse a few microseconds after the Y coordinate has been latched, to intensify the selected spot on the scope (all scopes have a "Z input" for that purpose). To draw a graph or set of characters on the screen, all you do is output successive XY coordinates repetitively (send X, then Y), fast enough so the eye doesn't see the flicker. Microcomputers are fast enough to display a few thousand XY pairs repetitively without annoying flicker. Given the fact that video (raster-scanned) displays for viewing are commonplace on microcomputers, this example might be more useful as an ultra-high-resolution plotter for photographic "hard copy," using 14-bit DACs and a micro-spot-size hard-copy scope display (see the next exercise).

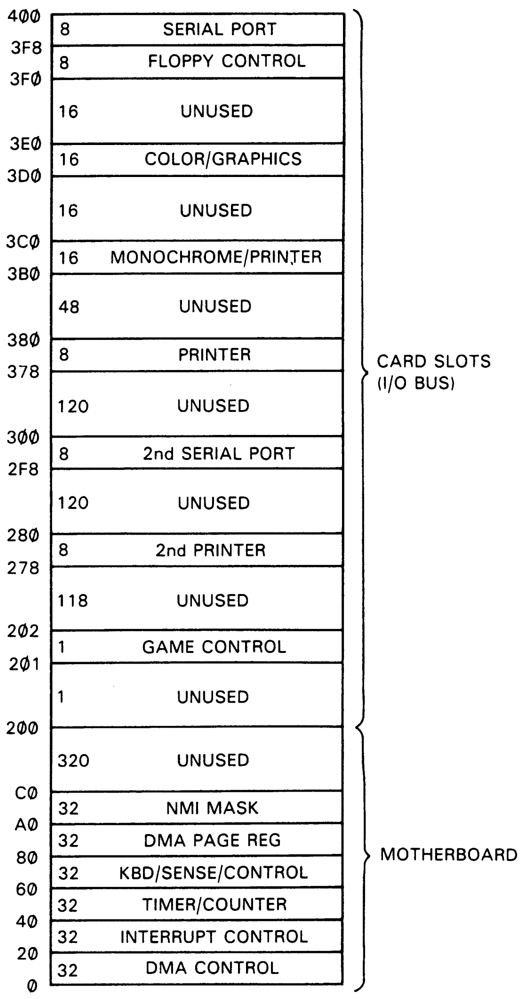


Figure 10.8. I/O addresses for IBM PC.

Some useful comments: (a) Note that we've arranged polarities so that the *trailing* edge of IOW′ clocks the *D* flip-flops; that is essential, since the data isn't yet valid on the leading edge of IOW′. If we were being very careful, we would check to see that required setup and hold times are satisfied for the '574s; in fact, for a slow bus like the PC's, you can't go wrong, since there are more than 500ns from valid data to trailing edge of IOW′. (b) You can save a few parts by using a strobed decoder in the address decoding circuitry, as indicated. Decoders like the '138 (3-line to 8-line) and '139 (dual 2-line to 4-line) include one or more enable inputs, and they are handy in this sort of application. (c) Note also that we could have combined the 3-input and 2-input NANDs into 4-input NANDs; we kept them separate only for clarity, decoding the addresses first, then AND-ing with the IOW′ strobe. (d) In fact, we could have ignored A1 entirely, and the circuit would work just the same! However, it would then respond also to addresses 3C2 and 3C3 (as X and Y, respectively), in effect "wasting" two I/O locations. In practice you often cheat in this way, incompletely decoding the address, because it saves parts (and there is plenty of I/O space, even if you waste some). In this example we could then have connected IOW′ where A1 is now connected, and omitted the 2-input NANDs entirely. (e) An interface like this is more flexible if its address can be set using a DIP switch (or DIP jumper block); then you can always make sure its address doesn't conflict with that of another interface you've got somewhere else. In this case the change is simple – replace the "hardwired" address lines to the comparator with eight lines that have switches to ground and pullups to +5 volts. (f) We used separate octal registers and DACs in this example for clarity. In real life you would probably choose a DAC with built-in latch (e.g., the "microprocessor-compatible" AD7528, a dual DAC with input latches); these even come in quad versions, (e.g., the AD7226) and in "double-buffered" versions with two cascaded latches for each DAC (e.g., the AD7225 quad).

EXERCISE 10.1
Redraw the address comparator logic with selectable I/O address.

EXERCISE 10.2
Redraw the XY display interface, using 16-bit DACs for both X and Y. You'll need four consecutive addresses: Assign the first two to the X register, and the last two to the Y register; use DIP-selectable I/O base address, of course. In each case the even address is the low-order byte, and the odd address is the high-order byte; that's the good choice, because that's how the 8086 stores 16-bit words, so you can use *word* I/O instructions to send data to your interface.

Programming the scope display

The programming to run this interface is straightforward. Program 10.3 shows what you do. The addresses of the first X and Y, and the number of points to be plotted, have to be available to the program. The display program will probably be a subroutine, with those parameters passed as arguments in the subroutine call. The program puts the addresses of the X and Y arrays (i.e., the address of the first X and Y) into address pointer registers SI and DI, and the byte count into CX. It then enters a loop in which successive XY pairs are sent to I/O ports 3C0 and 3C1. The X and Y pointers are advanced each time around, and the counter is decremented and tested for zero, which means the last point has been displayed; the pointers and counter are then reinitialized, and the process begins again.

A couple of important points: Once started, this program displays the XY array forever. In real life the program would probably check the keyboard to see if the operator wants the plot terminated. Alternatively, the display could be terminated after a specified time had elapsed, or by an "interrupt," which we will discuss shortly. With this sort of "refreshed" display, there usually isn't time to do much computing while displaying. A display device refreshed from its own memory takes that burden off the computer, and this is generally a better method. Nevertheless, if the objective is to make a precision plot for photographic hard copy, this program and interface (souped up as in Exercise 10.2) will do the job nicely.

Program 10.3

```
                                    ;routine to drive XY display
INIT:   MOV   SI,xpoint             ;initialize x pointer
        MOV   DI,ypoint             ;initialize y pointer
        MOV   CX,npoint             ;initialize counter

PLOT:   MOV   AL,[SI]               ;get x byte
        OUT   3C0H,AL               ;send it out
        MOV   AL,[DI]               ;get y byte
        OUT   3C1H,AL               ;send it out
        INC   SI                    ;advance x pointer
        INC   DI                    ;advance y pointer
        DEC   CX                    ;decrement counter
        JNZ   PLOT                  ;not done, plot more stuff
        JMP   INIT                  ;done, start over
```

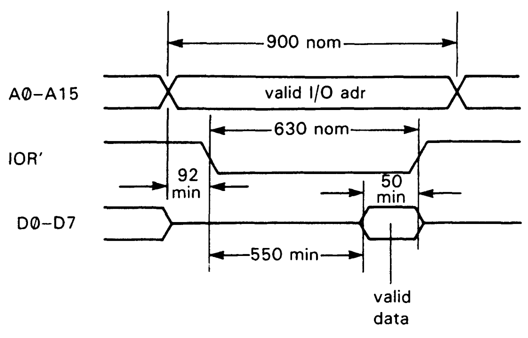


Figure 10.9. I/O READ cycle.

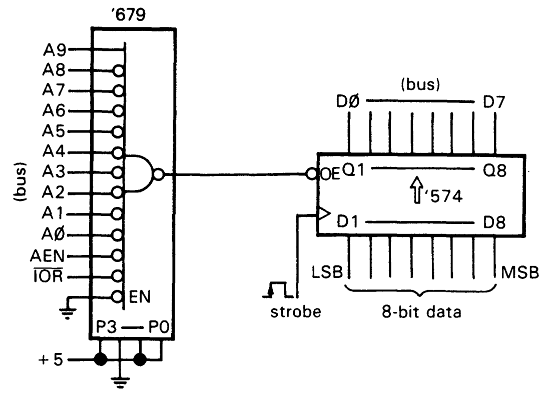


Figure 10.10. Parallel input port.

10.07 Programmed I/O: data in

The other direction of programmed I/O is equally simple. The interface looks at the ADDRESS lines as before. If it sees its own address (and AEN is LOW), it puts data onto the DATA lines coincident with the IOR′ pulse (Fig. 10.9). Figure 10.10 shows an example. This interface lets the PC read a byte latched in the '574 *D*-type register. Since the clock input and data inputs of the register are accessible to an external device, the register could hold just about any sort of digital information (the output of a digital instrument, A/D converter, etc.). For variety, we've eliminated all gates by using a '679 "12-bit address decoder" IC. It's a clever chip with 12 address inputs, an enable, and 4 "programming" inputs. If you want to decode a fixed address, it does the trick: It's functionally a 12-input NAND gate, for which a programmable number of the inputs can be inverted; the inverted inputs are always the lowest-numbered ones, and the number of them is the number you have asserted at the (4-bit) programming inputs.

In this case we've decided to plunder that lonely unused port at I/O address 200_H (Fig. 10.8). We need to recognize the state A9 = HIGH, A0–A8 = LOW. We might as well use the '679 to qualify the decoded address with AEN = LOW and IOR′ = LOW. So altogether we need a NAND with 11 inverted inputs and 1 noninverted input, which we get by hard-wiring the programming inputs with 11 in binary (1011). Then we connect the address lines and strobe as shown. When an

```
IN  AL,200H
```

instruction is executed, the CPU asserts 200_H on A0–A9, waits a while, then asserts IOR′ for 630ns. The CPU latches what it sees on the DATA bus (D0–D7) at the trailing edge of IOR′, then disasserts A0–A9. The peripheral's responsibility is to get the data onto D0–D7 at least 50ns before the end of IOR′; that's pretty relaxed timing, since it has known that data is being requested from it for at least 600ns. With typical HC or LS gate propagation times of 10ns, 600ns looks like forever.

Beginning with this example, we will omit the tangle of bus lines and simply call them out by name.

Bus signals: bidirectional versus one-way

From the two examples we've done so far you can see that some bus lines are *bidirectional*, for example the DATA lines: They are asserted by the CPU during write, but asserted by the peripheral during read. Both CPU and peripheral use three-state drivers for these lines. Others, like IOW′ and IOR′, are always driven by the CPU,

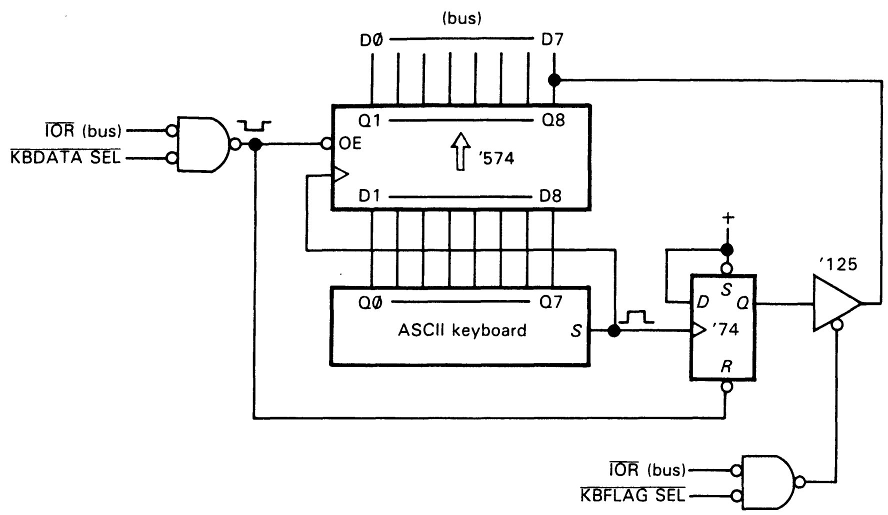


Figure 10.11. Keyboard interface with status bit.

with standard totem-pole driver chips. It is typical of computer buses to have both kinds of lines, using bidirectional lines for data that goes both ways, and one-way lines for signals that are always generated by the CPU (or, more accurately, generated by the associated bus control logic). There is always some clear protocol, like our rules for asserting/reading according to IOW′, IOR′, and ADDRESS, to prevent "bus contention" on these shared lines.

Of the signals so far, only the DATA lines are bidirectional; the ADDRESS lines, AEN, and strobes are one-way from the CPU. (Lest we give the wrong impression, we should point out that more complex computer systems permit other riders on the bus to become bus "masters"; obviously in such a system nearly all bus signals must be shared and bidirectional. The PC is unusually simple.)

10.08 Programmed I/O: status registers

In our last example, the computer can read a byte from the interface any time it wants to. That's nice, but how does it know when there's something worth reading? In some situations you may want the computer to read data at equally spaced intervals, as determined by its "real-time clock." Perhaps the computer instructs an A/D converter to begin conversions at regular intervals (via an OUT command), then reads the result a few microseconds later (via an IN command). That might suffice in a data-logging application. However, it is often the case that the external device has a mind of its own, and it would be nice if it could communicate what's happening to the computer without having to wait around.

A classic example is an alphanumeric input terminal, with someone banging away at a keyboard. You don't want characters to get lost; the computer has to get every character, and without much delay. With a fast storage device like disk or tape the situation is even more serious; data must be moved at rates up to 100,000 bytes per second without delay. There are actually three ways to handle this general problem: status registers, interrupts, and direct memory access. Let's begin with the simplest method – status registers – illustrated by the keyboard interface in Figure 10.11.

In this example, an ASCII keyboard drives a '574 octal *D*-type register, clocking in a character via the keyboard's STB (strobe) output pulse when a key is struck. We rig up the standard programmed data-incircuit, as shown, using the three-state outputs of the '574 to drive the DATA bus directly. The input labeled KBDATA SEL′ comes from an address decoding circuit of the sort shown explicitly in the previous examples, and it goes LOW when the particular address chosen for this interface appears on the ADDRESS lines of the bus (in combination with AEN asserted LOW).

What's new in this example is the flip-flop, which gets set when a character is struck, and cleared when a character is read by the computer. It's a 1-bit *status* register, HIGH if there's a new character available, LOW otherwise. The computer can query the status bit by doing a data IN from the other address of this device, decoded (with gates, decoders, or whatever) as KBFLAG SEL′. You need only one bit to convey the status information, so the interface drives only the most significant bit, in this case with a '125 three-state buffer. (*Never* drive a bidirectional line with a totem-pole output!) The line coming into the side of the buffer symbol is the three-state output enable, asserted when LOW, as indicated by the negation bubble.

Program example: keyboard terminal

The computer now has a way to find out when new data is ready. Program 10.4 shows how. This is a routine to get characters from keyboard terminal, whose data port address is KBDATA (it's good programming style to define the actual numerical port addresses - which correspond to what the hardware decodes as KBDATA SEL, etc. - in some statements near the beginning of the program, as shown); each character is "echoed" on the computer's display device (port address = OUTBYTE). When it has gotten a whole line, it transfers control to a line-handling routine, which might do just about anything, based on what the line says. When it's ready for another line, it types an asterisk. This sort of function should make sense to you if you've had some experience with computers.

The program begins by initializing the character buffer pointer, by moving the *address* of the buffer that we just allocated to the address register BP. Note we can't just say

```
MOV  BP,charbuf
```

because that would load the *contents* of charbuf, not its address; in 8086 assembly language you use the word "offset" in front of a memory label to signify its address. The program then reads the keyboard status bit via an IN instruction, ANDs it with 80_H to keep only the status bit (this is called "masking"), and tests for zero. Zero means the bit isn't set, so the program loops. When a nonzero status bit is detected, it reads the keyboard data port (which clears the status flag flip-flop), stores it consecutively in the line buffer, increments the pointer (BP), and calls the routine that echoes the character to the screen. Finally, it checks to see if the line was terminated by a carriage return: If it wasn't, it goes back and loops on the keyboard status flag again; if it was a CR, it transfers control to the line handler, after which it types an asterisk and begins the entire process anew.

A subroutine has been used to display a character, since even that simple operation requires some flag checking and masking. The routine first saves the byte into AH, then reads and masks the screen's busy flag. Nonzero means the screen is busy, so it keeps checking; otherwise it restores the character to AL, sends it to the screen's data port, and returns.

Some notes on the program: (a) We could have omitted the keyboard flag masking step, since the MSB (where we put the flag bit in our hardware) is the sign bit; thus we could have used the instruction JPL KFCHK. However, this trick works only for testing the MSB and thus is

Program 10.4

```
                                  ;keyboard handler -- uses flags
KBDATA  equ ***H                  ;put kbd data port adr here
KBFLAG  equ ***H                  ;ditto for kbd flag
KBMASK  equ 80H                   ;kbd flag mask
OUTBYTE equ ***H                  ;put disp port adr here
OUTFLAG equ ***H                  ;ditto for disp port flag
OUTMASK equ ***H                  ;disp port busy mask

charbuf  DB 100 dup(0)            ;allocates buffer of 100 bytes

INIT:  MOV  BP,offset charbuf ;initialize char buffer pointer
KFCHK: IN   AL,KBFLAG             ;read kbd flag
       AND  AL,KBMASK             ;mask unused bits
       JZ   KFCHK                 ;flag not set -- no data
       IN   AL,KBDATA             ;get new kbd byte
       MOV  [BP],AL               ;store it in line buffer
       INC  BP                    ;advance pointer
       CALL TYPE                  ;echo last char to display
       CMP  AL,0DH                ;was it carriage return?
       JNZ  KFCHK                 ;if not, get next char
LINE:    o                        ;if so, do something with line
         o                        ;keep at it
         o                        ;don't quit now
         o                        ;done at last!
       MOV  AL,'*'
       CALL TYPE                  ;type a "prompt" -- asterisk
       JMP  INIT                  ;get another line

                                  ;routine to type character
                                  ;types and preserves AL
TYPE:  MOV  AH,AL                 ;save the char in AH
PCHK:  IN   AL,OUTFLAG            ;check printer busy?
       AND  AL,OUTMASK            ;printer flag mask
       JNZ  PCHK                  ;if busy check again
       MOV  AL,AH                 ;restore char to AL
       OUT  OUTBYTE,AL            ;type it
       RET                        ;return
```

somewhat specialized. (b) In keeping with good programming practice, the carriage return symbol (0DH) and asterisk probably should be defined constants, similar to KBMASK. (c) The line handler probably should be a subroutine, also. (d) Characters will be lost if the line handler takes too long; this leads us to the more elegant approach of *interrupts*, which we'll take up shortly. (e) Keyboard and terminal handlers are used so often that the PC provides built-in handlers, accessed through "software interrupts" (we'll see them later); thus, our program isn't even needed!

Status bits generalized

This keyboard example illustrates status bit protocol; but it's so simple that you may come away with the wrong idea. In an actual peripheral interface of some complexity, there will usually be several flags to signal various conditions. For example, in a magnetic-tape interface you will normally have status bits for beginning of tape, end of reel, parity error, tape in motion, etc. The usual procedure is to put all the status bits into one byte or word, so that a data IN command from the status register gets all bits at once. Typically you would have a bit indicating any of a set of error conditions as the MSB of the status word, so a simple check of sign tells if there are *any* errors; if there are, you test specific bits of the word (by ANDing with masks) to find out what's wrong. Furthermore, in a complex interface you probably wouldn't have the status bits reset "automatically," as we did with our single bit; instead, a data OUT statement might be used, each bit of which clears a specific flag.

EXERCISE 10.3

With our keyboard interface there is no way for the computer to know if it missed a character. Modify the circuit so there are two status bits: CHAR READY (that's what we have already) and LOST DATA. The LOST DATA flag should be readable as D6 on the same status port as CHAR READY; it is 1 if a key was struck before the previous character was fetched by the computer, zero otherwise.

EXERCISE 10.4

Add a program segment to Program 10.4 that checks for lost data. It should call a subroutine called LOST if it detects lost data; otherwise it should work as before.

10.09 Interrupts

The use of status flags just illustrated is one of three ways for a peripheral device to "tell" the computer when some action needs to be taken. Although it will suffice in many simple situations, it has the serious drawback that the peripheral cannot "announce" that some action needs to be taken – it has to wait to be "asked" by the CPU, via a data IN command from its status register. Devices that need quick action (such as disks or latency-sensitive real-time I/O) would have to have their status flags queried often, and with a few such devices in a computer system the CPU would soon find itself spending most of its time checking status flags, as in the last example.

Furthermore, even with continual status flag checking you can still get in trouble: In the last example, for instance, the CPU will have no trouble keeping up with someone typing at the keyboard when it is in the main (flag checking) loop. But what if it spends 1/10 second in the line-handling portion? Or what if the display device is a slow one, making the program wait for its busy flag to clear?

What is needed is a mechanism for a peripheral to *interrupt* the normal action of the CPU when something needs to be done. The CPU can then check the status register to find out what the trouble is, take care of what needs to be done, and go back to its normal business.

To add interrupt capability to a computer, it is necessary to add a few new bus signals: At least one shared line for peripherals to signal an interrupt, and (usually) a pair of lines by which the CPU can determine who interrupted. As luck would have it, the IBM PC is not a very instructive example, because it does not implement a full interrupt capability. What it lacks in power, though, it more than makes up for in simplicity; implementing hardware interrupts in a PC peripheral interface is like falling off a log.

Here's how it works: The PC bus has a set of 6 *interrupt request* lines, called IRQ2–IRQ7. They are positive-true inputs to the CPU's support circuitry (specifically, to the 8259 interrupt controller). To make an interrupt, you simply bring one of the lines HIGH. If interrupts are enabled in general (along with the particular IRQ you assert), the CPU will break off after its next instruction, then (after saving its flags and current location onto the stack) jump to an "interrupt-handler" program somewhere in memory. You write the handler to do what you want (e.g., get keyboard data), and you can put the handler anywhere you wish, because the CPU figures out where to jump by

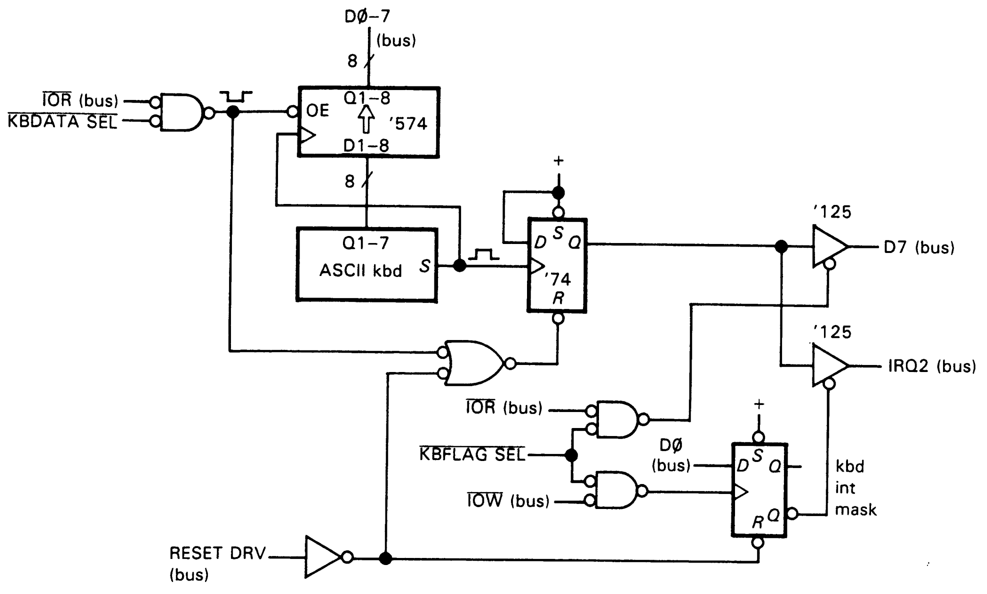


Figure 10.12. Keyboard interface with interrupts.

looking for the handler's 4-byte address in a special location in low memory. That location depends on which IRQ you've asserted; for the 8086 it is given in hex by 20 + 4n, where n is the interrupt level. For example, the CPU would respond to an interrupt on IRQ2 by jumping to the (4-byte) address stored in locations 28_H through $2B_H$ (it's just like indirect addressing, except that the address is found in memory rather than in a register); of course, you would have cleverly arranged for the starting address of your handler to be there. At the end of your handler you execute an IRET instruction, which causes the CPU to restore the preexisting flag register and jump back to wherever it was when the interrupt happened.

Let's illustrate by adding interrupts to the keyboard interface (Fig. 10.12). We've left the flag bit ("character ready") and programmed I/O circuitry essentially as before, except that we've ORed the flag clear with a new bus line, RESET DRV, a CPU output that is momentarily asserted HIGH when the computer is turned on. This signal is generally used to force your flip-flops and other sequential logic into a known state at power-up. Obviously it should reset a flag that indicates a valid byte is ready to be claimed (and that, in our new interface, will even cause an interrupt). The only other change we've made is to use a compact notation for the byte-wide data paths, to make the diagram easy to read.

The new interrupt circuitry consists of a driver to assert IRQ2 when a character is ready. That's all the new hardware you need. Although not strictly necessary, we've added the capability to disable the interrupt driver (it's a three-state buffer) by sending a byte with D0 LOW to the KBFLAG port address. This would be used if you wanted to plug in another peripheral with interrupts at the same IRQ level, allowing only one peripheral to use its interrupts at any given time (later we'll have further explanation on this awkward point).

10.10 Interrupt handling

The IBM PC/XT family makes interrupt handling easy (though limited in flexibility) by using an 8259 interrupt-controller IC on the motherboard. This chip does most of the hard work, which consists of prioritizing, masking, and asserting vectors (we'll describe these after finishing the example). The CPU, for its part, recognizes the interrupt and responds by saving the instruction pointer and flag register, disabling further interrupts, then making a jump via the corresponding address stored in the low-memory vector area. Your handler program does the rest, namely: (a) save (push) any registers you'll be using (remember that the interrupted program can't prepare for the interrupt, since it can happen anywhere in the running program; it's a bolt out of the blue), (b) figure out what needs to be done, by reading status register(s) if necessary, (c) do it, (d) restore the saved registers from the stack, (e) tell the 8259 you're done (by sending an "end-of-interrupt" byte 20_H to its register at I/O address 20_H), and finally (f) execute a return from interrupt instruction IRET; this causes the CPU to restore the old flag register that it saved on the stack, and jump (via the old instruction pointer it saved on the stack) back to the program that was interrupted. Somewhere in the program, you must have (g) loaded the handler's address into the vector location corresponding to the IRQ level used by the hardware, and told the 8259 to enable interrupts at that level.

Program 10.5 shows the code for the keyboard with interrupt. Here's the overall scheme: The main program sets things up, then loops on a flag (in memory, not hardware) that the interrupt handler sets when it recognizes a carriage return; when the main program sees that flag set, it goes off and does something with the line, then returns to the flag-checking loop. The handler, entered at each interrupt, puts a character into the line buffer, sets the flag if it was a carriage return, then returns.

Let's look at the program in some detail. After defining port addresses and the all-critical vector location for IRQ2, it allocates 100 bytes (initially filled with zeros) for the character buffer. The actual program execution begins by putting the buffer address in address register SI, zero in the end-of-line flag, and the address of the handler (which begins with KBINT) in location 28_H. To enable level-2 interrupts in the 8259, we clear bit 2 of its existing mask (IN, AND, OUT); then we enable CPU interrupts and send a 1 to KBFLAG, which enables the three-state driver. Now we're running. The program then loops, with interrupts secretly happening right under the main program's nose, until it mysteriously finds "buflg" set. It resets the pointer and flag immediately (in case another interrupt occurs soon), then gobbles up the line. It would be well advised either to move quickly or to copy the line to another buffer, since another interrupt (with a new byte to go in the buffer) could come along in a few milliseconds; in that time you can execute a few thousand instructions, however, more than enough to copy the line.

The interrupt handler is a separate little piece of code, with no entry from the main program. It gets entered upon a level-2 interrupt, via its address that we initially loaded into 28_H. It knows exactly what it has to do, and it does it without complaining: It saves AX (since it plans to clobber it), reads the character from the keyboard data port, puts it in the buffer, increments the pointer, echoes the character to the screen, sets the flag if it was a carriage return, sends end-of-interrupt to the 8259, restores AX, and returns.

If you look back at our list of handler tasks above, you'll see that we omitted just

Program 10.5

```
                                  ;keyboard handler -- uses interrupts
KBVECT equ word pntr 0028H        ;INT2 vector
KBDATA equ ***H                   ;put kbd data port adr here
KBFLAG equ ***H                   ;put kbd flag port adr here

buflg    DB   0                   ;allocates "buffer-full" flag
charbuf  DB 100 dup(0)            ;allocates 100-byte character buffer

       CLI                        ;disable interrupts
       MOV  SI,offset charbuf     ;initialize buffer pointer
       MOV  buflg,0               ;and end-of-line flag
       MOV  KBVECT,offset KBINT   ;handler adr to vector area
       IN   AL,21H                ;existing 8259 int mask
       AND  AL,0FBH               ;clear bit 2 to enable INT2
       OUT  21H,AL                ;and send to 8259 OCW1
       STI                        ;enable interrupts
       MOV  AL,1
       OUT  KBFLAG,AL             ;enable hardware 3-state driver

LNCHK: MOV  AL,buflg
       JZ   LNCHK                 ;loop until end-of-line flag set

LINE:  MOV  SI,offset charbuf     ;reset pointer
       MOV  buflg,0               ;and line flag
       MOV  AL,'*'
       CALL TYPE                  ;type prompt "*"
         o                        ;do something with line
         o
         o
       JMP  LNCHK                 ;and wait for another line

                                  ;keyboard interrupt handler
                                  ;an INT2 lands you here, via vector we loaded
KBINT: PUSH AX                    ;save AX register, used here
       IN   AL,KBDATA             ;get data byte from keyboard
       MOV  [SI],AL               ;put it in line buffer
       INC  SI                    ;and advance pointer
       CALL TYPE                  ;echo to screen
       CMP  AL,0DH                ;check for carriage return
       JNZ  HOME                  ;not a CR -- return
       MOV  buflg,0FFH            ;CR -- set end-of-line flag
HOME:  MOV  AL,20H
       OUT  20H,AL                ;end-of-interrupt signal to 8259
       POP  AX                    ;restore old AX
       IRET                       ;and return
```

one step, namely reading status flags to figure out which of several actions needed to be done. That's unnecessary here, though, because there's only one reason to interrupt, namely a new keyboard character needs to be read. (The programmer obviously has to understand under what conditions the hardware makes an interrupt, and what is required to service the interrupt.)

A few notes on this program: First, even though we're using interrupts, the program seems as dumb as before – it loops continually on the end-of-line flag.

However, it could be doing other things, if there were things to do. In fact, it does just that beginning at statement LINE, where it processes the finished line; during that time, interrupts make sure that new characters are put into the buffer, whereas they would have been lost in our previous example without interrupts.

This brings up a second point, namely, even with interrupts we're still in trouble if the program is doing things with the previous line when the next line has been completely entered. Of course, *on the average* the program simply has to keep up with keyboard entry; but you could have a situation in which the line user occasionally spends a lot of time, and you need to buffer more than one line temporarily. One solution to this is to make a copy to a second buffer, or to alternate between two buffers. An elegant alternative is to organize input as a queue, implemented as a "ring buffer" (or "circular buffer"), in which a pair of pointers keep track of where the next input character goes, and where the next character is removed. The interrupt handler advances the input pointer, and the line user advances the output pointer. Such a ring buffer might typically be 256 bytes long, permitting the line user to get behind by a few lines.

A third point concerns the interrupt handler itself. It's usually best to keep it short and simple, perhaps setting flags to signal the need for complicated operations in the main program. If the handler does become long-winded, you risk losing data from other interrupting devices, because interrupts are disabled when the CPU jumps into the handler. The solution in this case is to re-enable interrupts *within* your handler with an STI instruction, after doing the critical things that have to be done first. Then if an interrupt occurs, your interrupt handler will itself be interrupted. Since flags and return addresses are stored on the stack, the program will find its way back, first to your handler, finally to the main program.

10.11 Interrupts in general

Our keyboard example illustrates the essence of interrupts – a spontaneous hardware request for attention by a peripheral, producing a program jump to a dedicated handler routine (usually resulting in some programmed I/O), followed by a return to the code that was interrupted. Other examples of interrupting devices are real-time clocks, in which a periodic interrupt (often 10 per second, but 18.2 per second in the PC) signals a timekeeping routine to advance the current time; another example is a parallel printer port, which interrupts each time it is ready for a new character. By using interrupts, these peripherals let the computer interleave other tasks simultaneously; that's why you can be doing word processing while your PC is printing a file (and, of course, keeping proper time throughout).

The IBM PC does not, however, illustrate the full generality of interrupts. As we saw, it has a set of six IRQ lines on the bus, each one of which can be used only for a single interrupting device. The IRQ lines are numbered according to priority; in the event of multiple interrupts, the lowest-numbered interrupt is serviced first. Four of the IRQ lines are preassigned to essential peripherals, namely the serial port (IRQ4), hard disk (IRQ5), floppy disk (IRQ6), and printer port (IRQ7), leaving only IRQ2 and IRQ3 available [lines for two additional IRQ levels recognized in the IBM PC are not even brought out on the bus, being used on the motherboard for the 18.2Hz clock (IRQ0) and the keyboard (IRQ1)]. If you were to add a streaming tape backup or local area network, you would have to use IRQ2 and IRQ3. Furthermore, the interrupt is *edge-triggered*, which frustrates any reasonable possibility of using wired-OR to combine several peripherals on a single IRQ line.

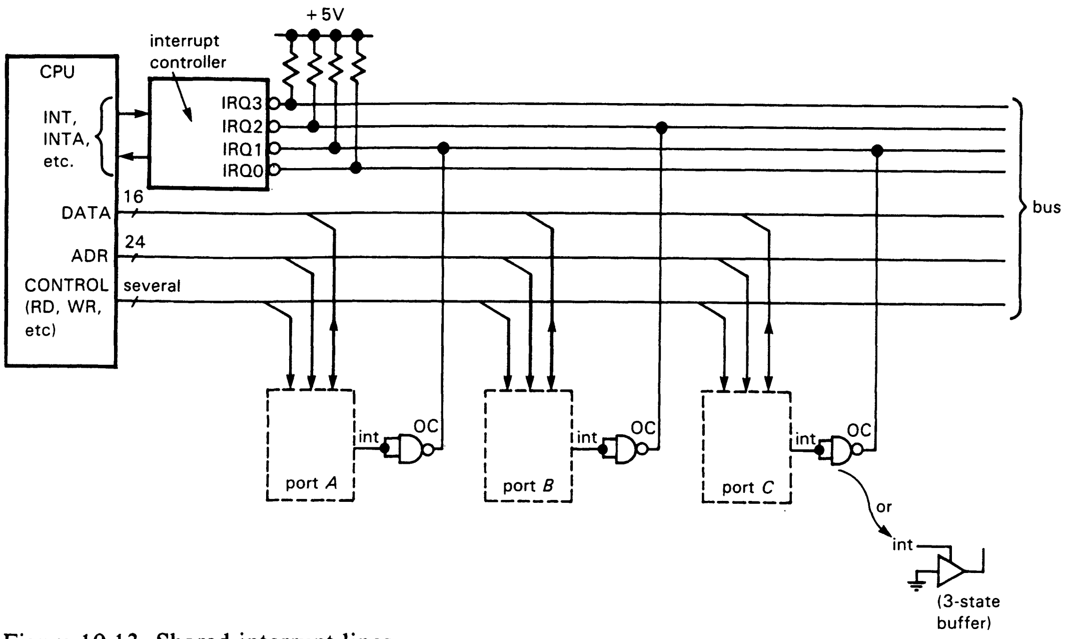


Figure 10.13. Shared interrupt lines.

□ *Shared interrupt lines*

The usual interrupt protocol, as implemented on many microcomputers, circumvents these limitations. Look at Figure 10.13. There are several (prioritized) IRQ-type lines; these are negative-true inputs to the CPU (or its immediate support circuitry). To request an interrupt, you pull one of the IRQ′ lines LOW, using an open-collector (or three-state) gate, as shown (note the trick for using a three-state gate to mimic an open-collector gate). The IRQ′ lines are shared, with a single resistive pullup, so you can put as many devices on each IRQ′ line as you want; in our example two ports share IRQ1. You would generally connect a latency-sensitive (impatient) device to a higher-priority IRQ′ line.

Since the IRQ′ lines are shared, there could always be another device interrupting on the same line at the same time. The CPU needs to know who interrupted, so it can jump to the appropriate handler. There is a simple way, and a complicated way, to do this. The simple way is called *autovectored polling* and is used nearly universally (though not on the IBM PC). Here's how it works.

□ *Autovectored polling.* Some circuitry on the CPU board (we'll have an example in Chapter 11) instructs the microprocessor that it is to use autovectoring, which works just like the IBM PC – each level of interrupt forces a jump through a corresponding vectoring location in low memory. For example, the 68000 microprocessor family we'll meet in Chapter 11 has seven levels of prioritized interrupt, which autovector through 4-byte pointers stored in the 28 (7 × 4) locations 64_H through $7F_H$. You put the addresses of the handlers in those locations, just as in our example above. For example, you would put the (4-byte) address of the handler for a level-3 interrupt in hex locations 6C through 6F.

Once in the handler, you know which level of interrupt you're servicing; you just don't know which particular device caused the interrupt. To find out, you simply check the status registers of each of the devices connected to that level of interrupt (a device *never* requests an interrupt without also indicating its need by setting one or more readable status bits). If a bit is set indicating that something needs to be done, you do it, including whatever it takes to cause the device to disassert its IRQ′: Some devices (like our keyboard) clear their interrupt when read, whereas others may need a particular byte sent to some I/O port address.

If the device you serviced was the only one interrupting at that level, that IRQ′ will now be HIGH upon returning to the interrupted program, and execution will continue. However, if there had been a second interrupting device at the same level, that IRQ′ line will still be held LOW (by the wired-OR action of the shared IRQ′ line) upon return from the service routine, so the CPU will immediately autovector back to the same handler. This time the polling will find the other interrupting device, do its thing, and return. Note that the order in which you poll status registers effectively sets up a "software priority," in addition to the hardware priority of the multiple IRQ′ levels.

□ *Interrupt acknowledgment.* We shouldn't leave the subject of interrupts without mentioning a more sophisticated procedure for identifying who interrupted – *interrupt acknowledgment.* In this method the CPU doesn't need to poll the status registers of possible interrupters, because the interrupting device *tells* the CPU its name, when asked. The interrupter does this by putting an "interrupt vector" (usually a unique 8-bit quantity) onto the DATA lines in response to an "interrupt acknowledge" signal that the CPU generates during the interrupt processing.

Nearly every microprocessor generates the needed signals. The sequence of events goes like this: (a) The CPU notices a pending interrupt. (b) The CPU finishes the current instruction, then asserts (i) bus signals that announce an interrupt, (ii) the interrupt level being serviced (on the low-order ADDRESS lines), and (iii) READ-like strobes that invite the interrupting device to identify itself. (c) The interrupting device responds to this bus activity by asserting its identity (interrupt vector) onto the DATA lines. (d) The CPU reads the vector and jumps into the corresponding unique handler for the interrupting device. (e) The handler software, as in our last example, reads flags, gets and sends data, etc., as needed; among its other duties, it must make sure the interrupting device disasserts its interrupt. (f) Finally, the interrupt handler software returns control to the program that was interrupted.

Sharp-eyed readers may have noticed a flaw in the procedure just outlined. In particular, there has to be a protocol to ensure that only one device asserts its vector, since there may be several simultaneous interrupting devices at the same IRQ level. The usual way to handle this is to have a bus signal (call it INTP, "interrupt priority") that is unusual in not being shared by devices on the bus, but rather is passed along *through* each device's interface circuit, beginning as a HIGH level at the device closest to the CPU and threading along through each interface. That's called a "daisy chain" in the colorful language of electronics. The rule for INTP hardware logic is as follows: If you have not requested an interrupt at the level being acknowledged, pass INTP through to the next device unchanged; if you *have* interrupted at that level, hold your INTP output LOW. Now the rule for asserting your vector goes like this: Put your vector number onto the data bus when requested by the CPU only if (a) you have an interrupt pending at the level being acknowledged and (b) your input INTP is HIGH. This guarantees that only one device asserts its vector; it also establishes a "serial priority" chain within each IRQ level, with devices electrically closest to the CPU getting serviced first. Computers that implement this scheme have little jumper plugs to pass INTP over unused motherboard slots. Don't forget to remove these jumpers when you plug in a new interface card (and put them back when you take one out!).

There is a nice alternative to the serial daisy-chain method of interrupt acknowledgment: Instead of threading a line through each possible interrupter, you bring individual lines back from each one to a priority encoder (Section 8.14), which in turn acknowledges the interrupt by asserting the identity of the highest-priority interrupting device. This scheme avoids the nuisance of daisy-chain jumpers. We will describe it in detail in Section 11.4 (Fig. 11.8).

In most microcomputer systems it isn't worth implementing the full-blown interrupt acknowledgment just described. After all, with 8-level autovectoring you can handle up to 8 interrupting devices without polling, and several times that number with polling. Only in large computer systems, in which you demand fast response with dozens of interrupting devices present, might you succumb to the complexity of the interrupt acknowledgment protocol, whether with serial daisy-chained hardware priority or with parallel priority encoding.

However, it is important to realize that even simple computers may be using vectored interrupt acknowledgment *internally*. For example, the simple 6-level autovector interrupt scheme of the IBM PC seen by the bus user is actually generated by an 8259 "programmable interrupt controller" chip that lives close to the CPU and generates the proper interrupt acknowledgment sequence just described (see below). This is necessary because the 8086 (and successors) can't implement autovectoring by themselves. On the other hand, the popular 68000 series of CPU chips can implement autovectoring internally, with just a single external gate (see Chapter 11).

□ *Interrupt masks*

We put a flip-flop in our simple keyboard example so that its interrupts could be disabled, even though the 8259 controller lets you turn off ("mask") each level of interrupt individually. We did that so that some other device could then use IRQ2. For a bus with shared (*level*-sensitive) IRQ′ lines, it is especially important to make each interrupt source maskable, again with an I/O output port bit. For example, a printer port normally interrupts each time its output buffer is empty ("give me more data"); when you've finished printing, though, you don't care. The obvious solution is to turn off printer interrupts. Since there might be other devices hooked to the same interrupt level, you must not mask that whole level; instead, you just send a bit to the printer port to disable its interrupts.

□ *How the IBM PC got the way it is*

The 8086/8 microprocessor used in the IBM PC actually implements the full vectored interrupt acknowledgment protocol. To keep things simple, however, the PC designers used an 8259 interrupt controller IC on the motherboard. The way it is used in the PC, it has a set of IRQ inputs from the I/O bus card slots (that's where you make your interrupt requests), and it connects to the microprocessor's data bus and signal lines. When it gets a request on an IRQ line from a peripheral, it figures out priority and goes through the whole business of asserting the corresponding vector onto the data bus. It has a mask register (accessible as I/O port 21_H) so that you can disable any specified group of interrupts.

The 8259 lets you select (through software) either *level-* or *edge*-triggered interrupts on its IRQ input lines, according to a byte sent to a control register (I/O port 20_H). Unfortunately, the PC designers decided to use edge triggering, probably because that makes it a little easier to implement interrupts (for example, you can just connect the real-time clock's square-wave output directly to IRQ0). If they had selected level-sensitive interrupts instead, you would be able to hang multiple interrupting devices on each IRQ′ line, with software polling as illustrated above. Unfortunately, the PC's ROM BIOS (Basic Input/Output System) and operating system (not to mention the hardware) assume edge triggering, so the choice is irrevocable. (Nearly all other computers, including even the successors to the PC and PC/AT, use level-sensitive interrupts.)

There is a partial solution to this problem. As long as there is an IRQ line available, you *can* combine several interrupting devices on a single PC board, with logic to generate edge-triggered interrupts on that single IRQ line; in fact, you could use your own 8259 (with its I/O ports accessible to the CPU) to do the job. But, since the interrupting devices have to know about each other, you can't use this scheme for independent plug-in peripherals. Furthermore, you still use up an IRQ line per card, and in a complicated system, given that there are only two IRQ levels free in the IBM PC, there will not be enough to go around.

Software interrupts

The Intel 8086-series of CPUs have an instruction ("INT n," where n is 0–255) that allows you to produce the same kind of vectored jump as an actual hardware interrupt. In fact, among its 256 possible jump vectors are duplicates of the 8 levels of IRQ-requested hardware interrupts (INT 8 through INT 15, to be exact). Thus, you can make a "software interrupt" from a program statement. The IBM PC uses these software interrupts to let you communicate with the operating system and its various ROM-based utilities. For example, INT 5 sends a replica of the screen to the printer. INT 21H turns out to be particularly important, because it is a function call to the operating system; you tell the system which DOS function you want by putting the corresponding number in register AH before executing the INT 21H.

Don't confuse these software interrupts with the externally triggered hardware interrupts we've been talking about. Software interrupts turn out to be a handy way of implementing vectored jumps from user code into system software. But they are not real interrupts, in the sense of a hardware call for attention from an external autonomous device. On the contrary, you build these into your software, you know when they are coming (that's why you can pass arguments through registers), and they are merely the response (albeit identical with what follows a true interrupt) of the CPU to its own code. You might think of software interrupts as a clever way to extend the instruction set.

10.12 Direct memory access

There are situations in which data must be moved very rapidly to or from a device. The classic examples are fast mass-storage devices like disk or tape, and on-line data-acquisition applications such as multichannel pulse-height analysis. Interrupt-initiated programmed processing of each data transfer in these examples would be awkward, and probably too slow. For example, data come from a "high-density" floppy disk at about 500kbit/s, or one byte every $16\mu s$. With all the bookkeeping involved in handling an interrupt, data would almost certainly be lost, even if the floppy were the only interrupting device in the system; with a few such devices the situation becomes hopeless. Even worse, a typical hard disk transfers a byte every 2μs, completely beyond the capability of programmed I/O. Devices like disks and tapes (not to mention real-time signals and data) can't stop in mid-stream, so a method must be provided for reliably fast response and high overall byte transfer rates. Even with peripherals with low average data transfer rates, there are sometimes requirements for short *latency* time, the time from initial request to actual movement of data.

The solution to these problems is direct memory access, or DMA, a method for direct communication from peripheral to memory. In some microcomputers (e.g., the IBM PC) the communication is actually handled by the CPU hardware, but that doesn't really matter. The important point is that no programming is involved in the actual transfer of data; bytes are moved between memory and peripheral via the bus, without program intervention. The only effect on the executing program is some slowing down of execution time, because DMA activity "steals" bus cycles that would otherwise be used to access memory for program execution. DMA usually involves more hardware complexity in the interface itself, and it should not be used unless necessary. However, it is useful to know what can be done, so we will describe briefly what you need to make a DMA interface. As with interrupts, the IBM PC designers streamlined their DMA protocol; a "DMA controller" chip on the motherboard does the hard work for you, making a DMA interface relatively straightforward. In general, though, DMA interfaces tend to be machine-dependent and complicated. We'll first explain the more usual "bus mastership" method of DMA, then the PC's simplified DMA protocol.

A typical DMA protocol

In DMA transfers, the peripheral requests access to the bus via special "bus request" lines (prioritized like the IRQ lines) that are part of the bus. The CPU gives permission and releases control of address, data, and strobing lines. The peripheral then asserts memory addresses onto the bus and either sends or receives data, one byte at a time, according to the strobes it asserts; in other words, it takes over the bus (it becomes "bus master") and acts like a CPU, directing data transfers. The DMA bus master is responsible for generating addresses (usually a block of successive addresses, generated with a binary counter) and keeping track of the number of bytes moved. The usual way to do this is to have a byte counter and an address counter in the interface. These are initially loaded from the CPU, via programmed I/O, to set up the DMA transfer desired. On command from the CPU (via a command bit, written with programmed I/O), the interface makes its DMA request and begins to move its data. It may release the bus between each byte (allowing the CPU to sneak a few instructions in), or it may take the more antisocial approach of keeping the bus for a block of transfers. When all transfers are complete, it releases the bus for the last time and notifies the program that it is finished by setting a status bit and requesting an interrupt, whereupon the CPU can decide what to do next.

Getting data or programs from disk is a common example of DMA transfer: The executing program asks for some "file" by name; the "operating system" (more about this soon) translates this into a set of programmed data OUT commands to the disk interface's control (or "command") register, byte count register, and address register (specifying where to go on the disk, how many bytes to read, and where to put them in memory). Then the disk interface finds the right place on the disk, makes a

DMA request, and begins moving blocks of data to the specified place in memory. When it's done, it sets bits in its status register to signify completion and then makes an interrupt. The CPU, which has meanwhile been executing other instructions (or possibly just waiting for data from the disk), responds to the interrupt, finds out from the status register of the disk interface that the data are now in memory, and then goes on to the next task. Thus, programmed I/O to the interface (the simplest kind of I/O) was used to set up the DMA transfer, DMA itself (stealing bus cycles from the CPU) was used for rapid transfer of data, and an interrupt was used to let the computer know the task was done. This sort of I/O hierarchy is extremely common, especially with mass-storage devices; you can expect maximum DMA transfer rates of one to ten million words per second on a typical microcomputer bus.

□ *DMA on the IBM PC*

The IBM PC, which is basically a simple microcomputer, has a simpler DMA protocol. The motherboard has a DMA controller (the Intel 8237) with built-in address and byte counters, along with the logic to disable the CPU and take over the bus. A peripheral that wants to do DMA, therefore, doesn't have to generate addresses and drive the bus. Instead, it signals the controller (via one of the three DRQ1–DRQ3 "DMA request" lines), which in turn responds by returning the corresponding DACK0–3′ ("DMA acknowledge"). The controller then controls the transfers, asserting address and strobing lines, with the peripheral asserting (or receiving) data to (or from) memory. In this whole process the memory sees nothing unusual going on, since addresses and memory strobes (MEMW′ or MEMR′), normally supplied by the CPU, are supplied by the 8237 controller, and if it's DMA *to* memory, data are supplied by the peripheral. The peripheral, on the other hand, knows something special is happening since it requested DMA access (and received confirmation via DACK′); so when the DMA controller asserts IOR′ (or IOW′), the peripheral supplies (or accepts) successive bytes. You might wonder why some innocent bystander peripheral doesn't get hurt in the DMA process, since both I/O strobes and addresses are being asserted, whereas the addresses are in fact the *memory* addresses that go with the memory strobes MEMW′ or MEMR′ asserted by the controller; they have nothing to do with I/O port addresses. The secret is our old friend AEN, specifically added to the bus just to solve this problem. AEN is asserted HIGH during DMA transfers, and all I/O port addressing must be qualified by ANDing with AEN LOW to prevent spurious responses to DMA memory addresses.

Even with the use of a separate controller chip, you still have to set up the starting address, byte count, and direction for the impending DMA transfer. These data go to the 8237, which is obliging, having a set of registers that you write (via programmed I/O) from the CPU. It's pretty straightforward (see Eggebrecht's book for clear guidance), except that, as with most peripheral LSI chips, there is a confusing variety of choices of "modes" (single transfer, block transfer, etc.). Luckily, the PC is sufficiently primitive that you're allowed to use only "single transfer," which transfers only one byte per DRQ request. If you insist on transferring a whole block of data by holding DRQ high, the 8237 releases the bus for one CPU cycle between each DMA cycle; that keeps the computer alive, even if you have a greedy peripheral that tries to hog the bus. The standard PC has a rather modest DMA capacity, about 2μs per byte transferred. As with interrupts, the PC is sparse on DMA channels: Three

TABLE 10.1. IBM PC BUS SIGNALS

Signal name	Number	Active	Type[a]	Direction CPU↔I/O	Pin #	Function
A0–A19	20	H	2S	→	A31-A12	address (A0-A15 for I/O)
D0–D7	8	H	3S	↔	A9-A2	data
IOR'	1	L	2S	→	B14	I/O read strobe
IOW'	1	L	2S	→	B13	I/O write strobe
MEMR'	1	L	2S	→	B12	memory read strobe
MEMW'	1	L	2S	→	B11	memory write strobe
AEN	1	H	2S	→	A11	DMA address signal
IRQ2–IRQ7	6	↑	2S	←	B4,B25-B21	interrupt request
RESET DRV	1	H	2S	→	B2	power-on reset
DRQ1–DRQ3	3	H	2S	←	B18,B6,B16	DMA request
DACK0'–DACK3'	4	L	2S	→	B19,B17,B26,B15	DMA acknowledge
ALE	1	H	2S	→	B28	"address latch enable"
CLK	1	-	2S	→	B20	CPU clock (4.77MHz)
I/O CH CK'	1	L	OC	←	A1	I/O error - makes NMI
I/O CH RDY	1	H	OC	←	A10	pull LOW for wait states
OSC	1	-	2S	→	B30	14.31818MHz (3xCPU clk)
T/C	1	H	2S	→	B27	DMA terminal count
GND	3	-	PS	→	B1,B10,B31	signal & power gnd
+5V DC	2	-	PS	→	B3,B29	+5V supply
+12V DC	1	-	PS	→	B9	+12V supply
–5V DC	1	-	PS	→	B5	-5V supply
–12V DC	1	-	PS	→	B7	-12V supply

(a) OC - open-collector; PS - power supply; 2S - 2-state (totem-pole); 3S - 3-state.

channels (DRQ1–DRQ3) are accessible on the I/O bus (DRQ0 is already used internally to refresh dynamic memory): DRQ1 is used for hard disk, and DRQ2 is used for floppy disk. That leaves DRQ3 for everything else.

10.13 Summary of the IBM PC's bus signals

Through our examples – programmed I/O, interrupts, and DMA – we've seen most of the bus signals that go to the card slots in the IBM PC. Table 10.1 (and Fig. 10.14) lists the full bus, with pin connections. For completeness we'll summarize them all here, beginning with the ones we've already met.

A0–A19

Address bus. Two-state, output only, active-HIGH. All 20 lines are used to address memory (with MEMR′ and MEMW′ as strobes, analogous to IOR′ and IOW′), but only the 16 least significant lines are used during I/O access

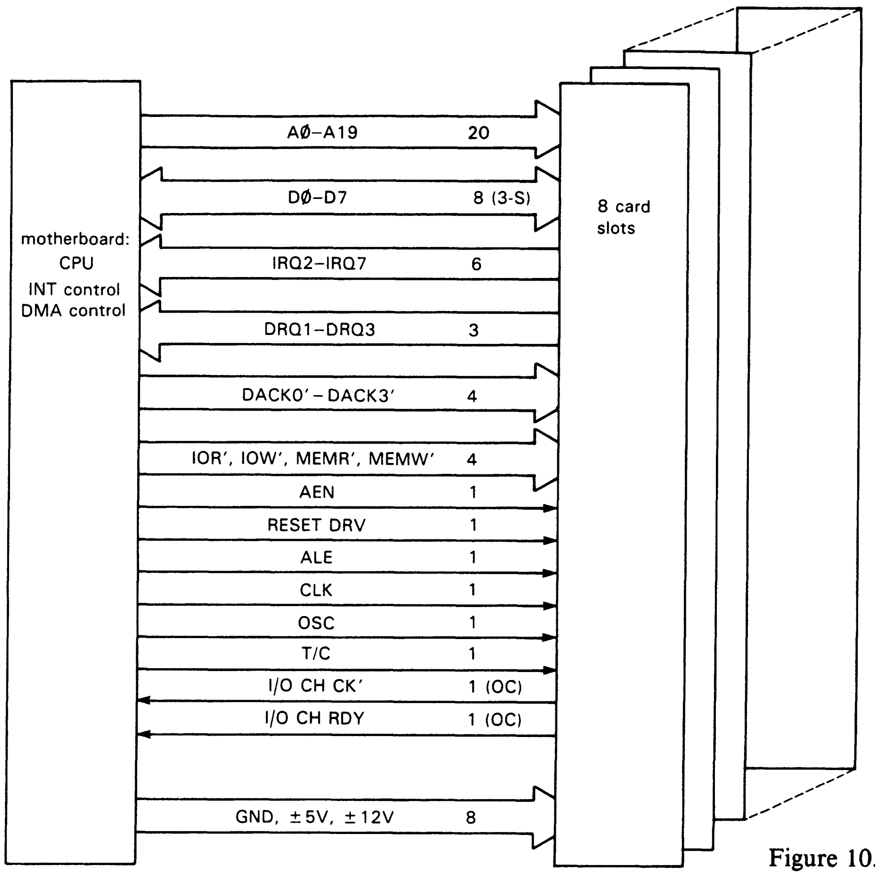


Figure 10.14. IBM PC bus.

(64K port addresses); I/O devices should qualify address with AEN LOW. Important note: I/O on the motherboard only looks at A0–A9, and uses 000_H–$1FF_H$; so external I/O must have its low ten bits in the range 200_H–$3FF_H$. You can be clever, though, by roosting in an unused 10-bit I/O address, then using the top 6 bits to create 64 I/O port addresses.

D0–D7

Data bus. Three-state, bidirectional, active-HIGH. Asserted by CPU during memory or I/O write; asserted by memory during memory read or DMA from memory; asserted by I/O port during I/O read or DMA to memory.

IOR′, IOW′, MEMR′, MEMW′

Data strobes. Two-state, output only, active-LOW. Asserted by CPU during read or write. On writes, data should be latched on trailing (rising) edge, qualified by address; on reads, data should be asserted coincident with strobe, qualified by address.

AEN

Address enable. Two-state, output only, active-HIGH. Asserted by CPU during DMA cycles. I/O ports must not respond with normal address decoding to IOR′ and IOW′; instead, I/O port that received DACK uses IOR′ or IOW′ to strobe DMA data bytes.

IRQ2–IRQ7

Interrupt request. Two-state, input only, rising-edge-triggered. Asserted by interrupting device. Prioritized, with IRQ2 highest, IRQ7 lowest. Maskable in 8259 interrupt controller, via CPU write to port 21_H. Each IRQ level can be used by only one device at a time.

RESET DRV

Reset driver. Two-state, output only, active-HIGH. Asserted by CPU during power-on. Used to initialize I/O devices to known start-up state.

DRQ1–DRQ3

DMA request. Two-state, input only, active-HIGH. Asserted by I/O device requesting DMA channel. Prioritized, with DRQ1 highest, DRQ3 lowest. Acknowledged by DACK1′-DACK3′.

DACK0′–DACK3′

DMA acknowledge. Two-state, output only, active-LOW. Asserted by CPU (DMA controller) to indicate grant of corresponding DMA request.

ALE

Address latch enable. Two-state, output only, active-HIGH. The 8088 used a multiplexed data/address bus, and this signal corresponds to the 8088's strobing signal, used by latches on the motherboard to latch the address. Can be used to signal beginning of a CPU cycle; usually ignored in I/O design.

CLK

Clock. Two-state, output only. This is the CPU's clocking signal; it's asymmetrical, 1/3 HIGH and 2/3 LOW. The original PC used a 4.77MHz clock, but higher speeds are common. CLK is used to synchronize wait-state requests (via I/O CH RDY), in order to stretch an I/O cycle for slow devices.

OSC

Oscillator. Two-state, output only. This is a 14.31818MHz square wave, which can be used (when divided by 4) as a color-burst oscillator for color display.

T/C

Terminal count. Two-state, output only, active-HIGH. Tells I/O port that a DMA block data transfer is complete. A DMA device must qualify it with DACK′ for the channel in use, since T/C is asserted when any of the DMA channels finishes a block transfer.

I/O CH CK′

I/O channel check. Open-collector, input only, active-LOW. Generates highest-priority interrupt (NMI, "nonmaskable interrupt"); used to signal error condition from some peripheral. CPU figures out who's in trouble by device polling (Section 10.11); each peripheral that can assert I/O CH CK′ must therefore have a status bit that can be read by the CPU.

I/O CH RDY

I/O channel ready. Open-collector, input only, active-HIGH. Generates "wait states" if disasserted (i.e., pulled LOW) before the second CLK rising edge of a processor cycle (normally 4 CLKs). Used to extend bus cycle for slow I/O or memory.

GND, +5VDC, −5VDC, +12VDC, −12VDC

Ground and dc supplies. Regulated dc voltages that are bused for use by peripheral interface cards. Check the specifications of your computer for power limitations, which are machine-dependent. Generally speaking, there should be enough power to run anything you can fit into the I/O slots.

□ 10.14 Synchronous versus asynchronous bus communication

The data IN/OUT protocol we described earlier constitutes a *synchronous* exchange of data; data are asserted onto or retrieved from the bus synchronously with strobing signals generated in the CPU (or DMA controller). Such a scheme has the virtue of simplicity, but it does open the possibility of problems with long buses, since the long propagation delays you get mean that data may not be asserted soon enough during a data IN operation for reliable transmission. In fact, with a synchronous bus the device sending the data never even knows if it was received! This sounds like a serious disadvantage, but in reality computer systems with synchronous buses seem to work just fine.

The alternative is an *asynchronous* bus, in which a data IN transfer, for example, goes something like this: The CPU asserts the port ADDRESS and a *level* (not a pulse) on a strobing line (call it IOR′, as before) that signifies data IN from the addressed device. The addressed device then asserts the DATA and a level signifying that the DATA is valid (call it DTACK′, "data transfer acknowledged"). When the CPU sees DTACK, it latches the DATA and then releases its IOR′ level. When the interface sees the IOR′ line go HIGH, it releases the DTACK′ and DATA lines. In other words, the CPU says "Give me data." The peripheral then says "OK, here it is." The CPU then says "OK, got it." And the peripheral finally says "Great! I'll go back to sleep again." This is sometimes referred to as "interlocked communication," or "handshaking."

Asynchronous bus protocol allows long buses and gives the communicating devices assurance that data are being moved. If a remote device is switched off, the CPU will know about it! Actually, that information is available via status registers with any kind of bus, and the chief advantage of asynchronous protocol is the flexibility of using any length of bus, bought at the expense of slightly greater hardware complexity.

There are situations where you want to attach relatively slow interface ICs to a bus; an example is slow-access ROM, or even RAM. All buses provide some way to stretch a bus cycle: With an asynchronous bus it's automatic, since the bus cycle goes on until the DTACK′ handshake is returned. With synchronous buses there is always some sort of HOLD′ line (it's called I/O CH RDY on the PC) to create wait states, effectively stretching the strobes and thereby delaying the end of the cycle. The overall bus cycle is always lengthened by an integral number of CPU clock cycles; that is the number of "wait states" inserted. For example, a standard IBM PC has a clock frequency of 4.77MHz (period of 210ns), and a standard memory access bus cycle is 4 clock periods (840ns). If I/O CH RDY is brought LOW before the second rising edge of CLK during a memory access, and brought HIGH again before the third, one wait state is generated, stretching the bus cycle (and MEMW′ or MEMR′) to 5 clocks (1050ns). If you hold I/O CH RDY LOW for additional clocks, you get additional wait states, up to a maximum of 10 clock periods.

Now we can reveal a well-kept secret about synchronous versus asynchronous buses: In reality, all single-processor (or, more precisely, single-bus-master) microcomputer buses are really synchronous, because all timing is slaved to a single CPU oscillator (like the 4.77MHz clock signal for the original IBM PC). Thus, if a peripheral device delays its handshake on an "asynchronous" bus, the cycle is always stretched by an integral number of CPU clocks. The distinction that is usually called synchronous versus asynchronous is really this: On an "asynchronous" bus, wait states are inserted by default unless a wired-OR line (DTACK′) is asserted LOW, whereas on a "synchronous" bus the default bus cycle has no wait states, which are generated only if a wired-OR line (HOLD′) is asserted LOW. The difference is more than semantic – you can't have a physically long bus with "synchronous" protocol, because the HOLD′ signal gets back too late to stretch the cycle, whereas with an "asynchronous" bus the CPU won't terminate the bus cycle without your permission (DTACK′). In our usually humble manner, we offer the following suggestive terminology to clear up this confusion: If the bus makes wait states by default ("asynchronous"), call it *default-wait*; if the bus makes wait states only when you ask ("synchronous"), call it *request-wait*. The IBM PC is request-wait, whereas the VME bus (see below) is default-wait.

This whole bus situation becomes more complicated with multiprocessor systems, in which bus mastership changes hands. A synchronous bus with multiple masters requires all masters to use a single clock, whereas an asynchronous bus permits different clock rates. Luckily for you, multiprocessor systems are beyond the scope of this book!

A possible point of confusion: You don't add wait states because you have a slow *peripheral* (a printer, for example); you do it only because you have a slow IC (ROM, say, with 250ns access time, or a slow LSI peripheral chip), which cannot latch (or produce) the data within the normal bus access time. A slow peripheral is usually hopelessly slow (*milli*seconds, not nanoseconds); the solution is to send (or receive) a byte at full bus speed, latching it in a byte-wide register chip, then wait for an interrupt (or possibly a status flag) before doing another full-speed transfer.

10.15 Other microcomputer buses

We chose the IBM PC to illustrate microcomputer bus architecture – bus signals, memory and programmed I/O, interrupts, and DMA. Since the PC is widely copied and widely used in engineering and data acquisition/control, this is a good illustrative choice for an electronics book. Furthermore, the PC bus is extraordinarily simple and easy to use.

However, simplicity has its costs. The original PC bus is seriously limited in important ways, some of which we have already mentioned (e.g., scarcity of interrupt and DMA channels). Even more seriously, by today's standards the PC bus has too little address space (20 bits, only 640K usable), too narrow a data path (8 bits), insufficient data transfer rate (1.2Mbyte/s, max), and no provision for multiple bus masters. IBM has evolved improved buses in subsequent PC generations, first the PC/AT (a compatible enhancement of the original PC), then the all-new (= incompatible!) "microchannel" bus of the PS/2 series. Outside the IBM world there are competing buses peculiar to a given manufacturer (e.g., DEC's Q-bus and VAXBI bus), and generic buses (Multibus, NuBus, VME bus). Let's take a quick tour of the computer buses listed in Table 10.2.

PC/AT and Micro Channel

IBM's PC/AT (for "Advanced Technology") was introduced in 1984, and discontinued in 1987, at the peak of its popularity, to make way for IBM's PS/2 series of "clone-killer" computers, which use the improved "Micro Channel" bus. [The PC/AT continued to thrive, however, since the clone manufacturers (and many buyers) initially ignored IBM's newer machines, whose advanced features required nonexistent software.] The PC/AT uses the 80286 CPU and a compatible

TABLE 10.2. COMPUTER BUSES

Bus	Raw bandwidth (Mbyte/s)	Data width	Address width	Block xfer?	MUXed data/adr?	Multimaster?	Sync/Async	IRQ lines[a]	Drivers	Connector[b]	Comments
STD bus		8	16	–	–	–	S	1	TTL	CE	controller-type applications
PC/XT	1.2	8	20	–	–	–	S	5E	TTL	CE	original IBM PC & compatibles
PC/AT	5.3	8,16	20,24	–	–	(c)	S	10E	TTL	CE	accepts PC/XT cards
EISA	33	8,16,32	20,24,32	•	–	•	S	11P	TTL	CE	enhanced PC/AT; auto-configure
MicroChannel	20	8,16,(32)	24,(32)	•	–	•	A	11	TTL	CE	IBM PS/2; auto-configure
Q-bus	2	16	22	•	•	•	A	4	(d)	CE	LSI-11, μVAX-I,II; daisy-chained IACK
Multibus I	10	8,16	20,24	–	–	•	A	8	TTL	CE	Intel; SUN-I and others
CAMAC	3	24	9	•	–	–	S	L	TTL/OC	CE	data acquisition & control bus
VAX BI	13.3	8,16,24,32	32	•	•	•	S	4	TTL	ZIF	VAX 780, 8600 series; parity
Multibus II	40	8,16,24,32	16,32	•	•	•	S	M	TTL	DIN	parity; 40MB/s for blk xfer, 20M otherwise
NuBus	40	32	32	•	•	•	S	M	TTL	DIN	Macintosh II adds 1 dedicated INT per slot; ""
VME	40	8,16,32	16,24,32	•	–	•	A	7	TTL	DIN	daisy-chained IACK; SUN-3
Futurebus	120					•	A	–	(d)		
Fastbus	160	32	32	•	•	•	A	M	ECL	H	communication across many crates

(a) E - edge-sensitive; L - LAM ("look at me"); M - "int" via bus mastership; P - programmable edge- or level-sensitive interrupts.
(b) CE - card-edge; DIN - 2-part "Eurocard" 96-pin connector; H - high density 2-part conn. (c) almost. (d) National Semi special.

enhancement of the original PC bus: An additional (and optional) connector carries an extra 8 bits of data, 4 bits of address, and 5 additional IRQ lines (edge-triggered, as before). The resulting 16-bit data path and higher CPU clock speed raise the maximum bandwidth to 5.3Mbyte/s, which, with the additional address space and interrupts, makes the PC/AT a serious microcomputer. The PC/AT bus (sometimes called Industry Standard Architecture, or ISA) even supports multiple bus masters, though its abilities here are limited. Cards that work on the original PC bus will work in the PC/AT (if they are fast enough), because you can ignore the bus enhancement carried on the extra connector; in that case, of course, you revert to an 8-bit data path and 20-bit address space. AT-compatible computers generally run their I/O bus at higher speeds, which can create additional timing problems with older plug-in boards.

The Micro Channel bus was first used in IBM's PS/2 series of second-generation personal computers, introduced in 1987. It allows for data and address paths up to 32 bits wide (in the high-end 80386-based machines), 11 levels of shared (level-sensitive) interrupts, multiple bus masters, and asynchronous protocol. Cards that plug into the Micro Channel don't have hardwired I/O port addresses; instead, the CPU assigns an address (and other configuration choices) at start-up, based on information it reads from ROM on the card. This pleasant feature means that you don't have to set little switches on each card, and you don't have to worry about cards using overlapping address space. Micro Channel cards have tight dimensional tolerances, owing to the daring use of 0.050 inch spacing between pads on the edge connectors.

EISA

The Extended Industry Standard Architecture (EISA) is the clone-makers' answer to the Micro Channel. It was introduced in 1988 by nine manufacturers of AT-compatible computers. By adding an extra connector to the AT bus, the EISA's designers implemented many of the desirable features of the Micro Channel, *while maintaining compatibility with existing AT plug-in cards.* Thus, you can plug standard AT boards into EISA and get normal AT functionality. Moreover, when used with boards designed specifically for it, the EISA supports 32-bit data transfers (with peak transfer rates of 33Mbyte/s), 32-bit memory addressing, multiple bus masters, programmable level or edge-triggered interrupts, and automatic board configuration.

Multibus I and II

Originally introduced by Intel, the Multibus formats have found their way into many computers. The original Multibus I is a capable bus with 16-bit data path and 24-bit address space, and it allows multiple bus masters. Multibus II is intended for high-performance multiprocessor systems, with 32-bit data and address paths, parity checking, distributed arbitration, and message-passing protocols. It uses a synchronous 10MHz clock and can transfer up to 40Mbyte/s to sequential addresses in "block transfer" mode. In common with some other large buses (NuBus, Fastbus), Multibus II saves pins by multiplexing data and address on a common set of 32 lines. It also uses a 96-pin card-mounted DIN connector, rather than the simple gold-plated "card-edge" connector: By using a well-designed card-mounted ("2-part") connector, you get better reliability and a connection system that is insensitive to card warp and rough handling.

Although Multibus II seems to have all the advantages, its flexibility can make your work hard. For example, it doesn't have conventional interrupts; instead, you "interrupt" by requesting bus mastership, then sending a message to the processor you want to interrupt! For simple systems, the simpler Multibus I (or some other simple bus) may be better.

NuBus

This is another high-performance synchronous multiprocessor bus with multiplexed 32-bit data and address paths, DIN connectors, and high data-transfer rates (to 40Mbyte/s in "block transfer" mode). In common with Multibus II, it forces you to go through a bus mastership protocol to interrupt. It is used in the high-end Macintosh computers where, thankfully, Apple added a dedicated interrupt line to each slot. Thus, each card slot has a unique vector assigned; the corresponding software handler knows which card interrupted without polling and has to poll only if that card has more than one possible interrupting device.

VME bus

The VME bus, like NuBus and Multibus II, is intended for multiprocessor 32-bit systems. Unlike those buses, however, it does not use multiplexed data/address lines. Nor does it use a synchronous master clock, preferring asynchronous protocol; this lets you mix processors of varying speeds without pain. The VME bus also implements conventional multilevel IRQ-type interrupts, with full interrupt acknowledgment (complete with daisy-chained INTP line). The VME bus is often viewed as an alternative to Multibus; for example, the original Sun computer from Sun Microsystems used Multibus, whereas their more recent Sun 2 and Sun 3 use VME. VME bus and Multibus II are currently slugging it out in the trade press, cheered on by Motorola and Intel, complete with diatribes and name-calling.

Fastbus and Futurebus

These are *very* high performance buses, with blazing speed. The Fastbus uses large cards (14×16 inches), ECL drivers, and arbitration protocols to support multiple bus masters. In fact, bus communication is one of its strong points, with capability for sophisticated "geographic" communication beyond the immediate crate of cards.

Q-bus and VAXBI

These are proprietary buses used in DEC computers. The Q-bus, used in the LSI-11 and early MicroVAX computers, evolved from DEC's original PDP-11 "Unibus." It supports 16-bit data and 22-bit addressing, asynchronous protocol with multiple masters, and multilevel IRQ-type interrupts. The VAXBI is a high-performance multiplexed 32-bit data/address bus used in the larger VAX 8600-series machines.

10.16 Connecting peripherals to the computer

Interfaces are usually built on printed-circuit cards or Wire-Wrap cards (see Chapter 12) designed to plug into the microcomputer's card slots. Microcomputers generally contain a number of unused slots for just this purpose (or they can be "expanded" to accommodate extra cards), with power-supply voltages and bus signals distributed to the card slots. Some machines use a "proprietary" bus (e.g., the IBM PC), others use a standardized microcomputer bus (e.g., the Sun 3 workstation, which rides on the VME bus), and some have no bus slots at all (e.g., the original Macintosh). Each bus has a standard card size (or sizes), ranging from the small 3.2×11.5 inch IBM PS/2 cards to the giant 14.4×15.9 inch Fastbus cards. Depending on the particular bus, each card has 50 to 300 connections along one edge, either in the form of a set of gold-plated printed-circuit edge connections or as a set of multipin connectors that are soldered to the board; the latter are known as "two-part" connectors and are generally more reliable than PCB edge connectors.

Commercially available interfaces for common tasks (disk, graphics, communications, analog I/O) are usually built on cards that plug into unused bus slots. Cables then go from connectors on the interface card to the peripheral (if any); if the interface involves many inputs or outputs (e.g., a digital logic analyzer), it may connect by cable to an external panel or box where there is more room for connectors (and additional circuitry). In either case it is common to use flat *ribbon cable*, with some care being taken to prevent cross-coupling of strobing signals with data. One method is to ground every other wire in the ribbon; another technique uses ribbon cable bonded to a flexible metal groundplane to reduce inductance and coupling, at the same time maintaining a nearly constant cable impedance. In both cases you can get nice multipin "mass-termination" connectors that attach to the cable with one simple crimping operation; check the catalogs of AMP, Berg, T& B Ansley, 3M, etc. An alternative to ribbon cable is a cable made of multiple twisted-pairs, each pair consisting of one signal line and one ground wire. Twisted-pair cable is available in many configurations, including a nifty ribbonlike flat cable (Allied/Spectra "Twist-'n-flat") in which there is a flat untwisted region every 20 inches for easy connection to crimp-on connectors of the type used for ordinary ribbon cable. Because of the strobed data-transfer protocol used between an interface card and the device it controls, it generally isn't necessary to use signal/ground pairs for *all* signal lines, just for the synchronizing pulses and other strobing or enabling lines. Suitable terminations and driver/receiver combinations should be used for long lines, as described in Section 9.14.

Custom interfaces are best handled in the same way, either laying out printed-circuit boards or using one of the general-purpose interfacing cards available commercially from companies such as Douglas, Electronic Solutions, and Vector. These blank cards have places for ICs and other components (including mass-termination connectors for external cables), and they come in solder and Wire-Wrap styles (more in Chapter 12). Some of them include built-in circuitry to handle bus communication, including interrupts and even DMA.

In some cases the best plan may be to build an interface that resides partly in the computer and partly outside, as suggested in Figure 10.15. In such cases the interface circuitry that goes in the computer will probably be a simple parallel input/output port, either a commercially available parallel port card or a custom card you design. The cable connecting the two parts of the interface is simple and could use one of the high-performance driver/receiver combinations we discussed in Section 9.14 if high-speed communication over long cable runs is needed (for example, RS-422, or the differential current-sinking 75S110 IC, or even fiber optics). This sort of scheme may be particularly useful for interfaces that handle low-level analog signals, since the noise-susceptible linear circuitry can be kept away from the general roar of digital interference present in the computer (and close to its analog signal source); this also allows you to pay careful attention to maintaining clean analog signal ground lines.

SCSI, IEEE-488, and other interfaces

There are literally hundreds of plug-in boards, performing an incredible variety of functions, available for common buses

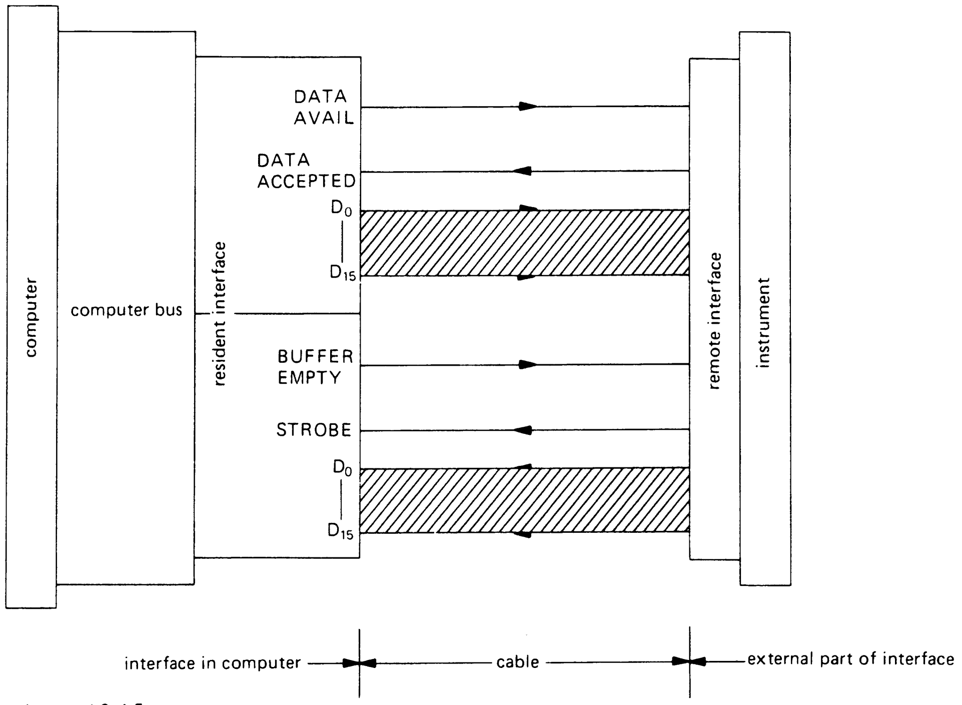


Figure 10.15

such as the IBM PC, Multibus, VME, and Q-bus. These are so inexpensive and easy to use that you should always check out the possibility that either (a) the board you are designing already exists or (b) you can use a simple parallel-port card as a computer-resident part of your interface, as described in the last section. Another possibility is to use a standard built-in "Centronics" parallel port, or an RS-232 serial port (see Sections 10.19 and 10.20), to couple a custom gadget to a microcomputer. This scheme has the virtue of making your gadget portable, even to a microcomputer with a different bus (or no bus at all!), since these ports look the same on all computers. Such a gadget for connection to a serial port will probably have its own microprocessor, so you might tend to think of it as a computer rather than a peripheral. But, as we'll explain in the next chapter, it's fun and easy (and cheap) to build little microprocessor-based instruments; there's really no good reason to treat a microprocessor differently from any other LSI chip, which you wouldn't hesitate to include in a custom circuit.

Following this idea a step further, there are a few "cable interface" standards that have become popular recently. They have names like SCSI ("small computer system interface"), IPI ("intelligent peripherals interface"), ESDI ("enhanced small-disk interface"), and IEEE-488 (also known as HPIB and GPIB, "general-purpose interface bus"). SCSI (pronounced "skuzzy") in particular is now standard equipment on many microcomputers, thanks to the availability of disks and other peripherals that connect directly to a SCSI port. You can get add-in SCSI interface cards for computers without built-in SCSI ports. SCSI is actually a descendant of SASI (Shugart Associates System Interface, a simple parallel bus that Shugart cooked up for their hard-disk drives) and in its simplest incarnation is a byte-wide bidirectional parallel protocol with handshaking. It allows several modes, including synchronous or asynchronous transfer, with single-ended or differential drivers; although it originally was used to connect a single CPU to a single disk, it can be used to couple multiple CPUs to multiple disks. Typical transfer rates are 1.5Mbyte/s (asynch) and 4Mbyte/s (synch); asynchronous protocol is slower because the handshakes are bouncing back and forth during each transfer. SCSI can go 20 feet with single-ended drivers, and 80 feet with differential drivers.

The IEEE-488 bus, originated by Hewlett-Packard as the HPIB, was designed for connecting laboratory instruments to computers. There is a full protocol for connecting multiple instruments on a bus, with phrases like "talkers" and "listeners." IEEE-488 is firmly entrenched in the instrumentation field, with manufacturers like Hewlett-Packard, Keithley, Philips/Fluke, Tektronix, and Wavetek offering it on most of their instrumentation. You can get 488 interfaces for nearly all microcomputers. We'll have more to say on SCSI and IEEE-488 in Section 10.20.

SOFTWARE SYSTEM CONCEPTS

In this section we will discuss some general aspects of small-computer programming, since a knowledge of computer interfacing is of limited value without an understanding of the hierarchy of programs that actually make the computer come to life. In particular, we would like to discuss the important areas of programming, operating systems, files, and use of memory. It is easy to get carried away admiring the beauty of computer hardware and underestimate the importance of good software. Software is what makes the computer fly, and a good operating system and package of "utilities" can make all the difference.

Following our discussion of software and systems, we will end the chapter with a section on communications concepts, in particular the standardized RS-232 serial ASCII protocol, the "Centronics" parallel port, other parallel communications schemes (SCSI, IPI, GPIB), and local area networks.

10.17 Programming

Assembly language

As we mentioned earlier in the chapter, the computer's CPU recognizes certain groupings of bits as valid instructions, which it then acts upon. It is extremely rare to program directly in this binary machine language. Instead, you write programs in a mnemonic assembly language (like our interfacing examples earlier), which a program called an *assembler* converts into relocatable machine code. Assembly language is very close to machine language; each instruction is converted directly into one line or a few lines of machine code (the first line is usually the operation code, with the extra lines generally completing the addressing of the variables, or furnishing constants). Assembly-language programming produces the most efficient code and allows you to get at flags and registers that are inaccessible from higher-level languages. But it is tedious programming, as the examples illustrate, and for most computing jobs (especially those involving plenty of numerical computation) it pays to use a compiled or interpreted high-level language, such as C or FORTRAN, with calls to assembly-language routines only where necessary.

Compilers and interpreters

C, FORTRAN, PASCAL, and BASIC are popular examples of high-level languages. You write a program with algebraic types of commands, for instance

$$x = (-b + sqrt(b * b - 4 * a * c))/(2 * a)$$

and with control structures like *if* ... *elseif* ... *else*, *for* ... , *while* ... , and *do* ... . You don't have to shuttle your little bytes hither and thither, or worry about addressing, saving registers, etc.; you just declare variables and arrays by type and size and use them in arithmetic or logical expressions. Everything is chocolate-coated.

This is called *source* code, from which there are two routes to a running program. Languages like C and FORTRAN are *compiled*, a process in which a language compiler converts the source-code statements to assembly code; from there it's business as usual, with the assembler converting that intermediate assembly language into machine language. Languages like BASIC and APL have traditionally been *interpreted*; instead of compiling an assembly-language program from the source program, an *interpreter* program "looks at" the statements and executes appropriate computer instructions.

In general, interpreted languages run much more slowly than compiled languages. However, since there's no compilation, assembly, or linking (discussed next), there's no delay after entering a program before it can run. Interpreters often include a simple editor, convenient for quick modification and retrial of a program you're debugging. Interpreted BASIC gained popularity in the early days of microcomputers, when hard disks were a rarity, since it ran entirely in memory; this contrasted with the tedious multipass compilation process. With today's fast disks and efficient compilers, there's not much to complain about. In fact, recent compilers have followed the lead of Borland's interpreted "Turbo Pascal" by providing a "total environment" in which you can hop around effortlessly between editor and running program: If there's a bug, the system puts you back into the editor, pointing to the bad statement; these compilers include debuggers, provision for making "libraries," and other pleasant features.

The current all-around favorite among heavy-duty programmers seems to be C, which combines the power of high-level languages with the beauty of structured languages and the bit-pushing flexibility of assembly code. However, FORTRAN still claims the lion's share of scientific computing.

Linkers and libraries

The assembler produces machine code (well, almost; it's actually called "relocatable machine code") from the assembly code produced by the compiler and from separate subroutines written in (or compiled to) assembly code. In addition, there are usually routines needed by particular commands in the high-level program. For example, a C program might need a math function like *sqrt*, or a host of I/O functions like *printf* or *fopen*. A program called a "linker" handles the bureaucratic nightmare of getting the appropriate subroutines (in relocatable form) from a "library," then rigging up all the linking jumps and addressing so the whole mess fits together in memory. It is the linker's job to put final numerical values into the memory references and variable addresses of the assembled code, and it can do this only when it knows which program calls which, and how long each program is. That's why the code produced by the assembler must be in relocatable form, as must the assembled subroutines that sit in the various libraries [there are usually several – a library of compiler functions, an I/O library, a math library, a library of system calls, and perhaps a home-grown (or store-bought) library of useful subroutines].

Editors and formatters

In prehistoric days (before 1970) you could find card-carrying computer programmers, literally: You wrote your programs by hand on coding forms, then punched them (or paid someone else to punch them) onto those handsome "IBM cards" that had rows of numbers printed on pastel cardboard. Nowadays even toddlers know how to use computer editors, the universal program entry method. Old-timers (those over 30) can still remember the first awkward "interactive" computer editors, with which you could create and modify a text file that, for some reason, the editor never let you see much of. Don Lancaster teased us with his "TV Typewriter," a build-it-yourself project that let you display a line of text on a television. That's all it did. No editing, no storage, no nothing. Our joy was truly unbounded, therefore, when we first used "full-screen" editors.

A good editor (and they're all good, now) lets you type and correct as you go, search for words, change text, move blocks of text around, open multiple windows on multiple files, and write "macro" definitions that do complex manipulations. The screen should redraw quickly, even if you add text near the beginning of a large file. Very large files shouldn't slow things down.

A general-purpose editor doesn't know, or care, what you are writing; it could be a program, a sonnet, or a book. It just creates the text file according to your keyboard instructions. If the file consists of statements in a programming language, the compiler, interpreter, or assembler reads it directly. If, on the other hand, the file is text that you want to print, you have two choices: You can send it directly to a printer, or you can mark it up with formatting information and send it to a formatter program that tells the printer how it should be printed. A good text formatter takes care of margins and line justification, proportional spacing, changes of font, italics, boldface, underline, and so on. The editor and formatter are often combined, sometimes with a screen display showing what the printed page will look like (that's called WYSIWYG, pronounced "wizzywig": what you see is what you get), but more often with the screen display only partially faithful to the final page. The most advanced formatters are capable of typesetting mathematical and scientific formulas. For "camera-ready" quality, you do your printing on a typesetting machine, which exposes photographic paper or film directly; laser or LED printers offer quite good quality at moderate cost and high speed; "dot-matrix" impact printers are the cheapest, as the result shows.

Editor/formatters go by names like MacWrite, Manuscript, Microsoft Word, Sprint, and WordPerfect. Popular technical formatters (which do both text and equations) are $\TeX$ and Troff. One caution: When creating text (as opposed to programs), most editor/formatters insert unusual characters in the edited text stream, for example to indicate italics, or temporary end-of-line. These characters are unacceptable to compilers and assemblers. Thus, you've got to force the editor to run in a "vanilla" mode, in order to create unadorned source code that the compiler, etc., won't choke on.

Here's some free advice: (a) Find a good editor and stick with it, and (b) Don't try to persuade others that your editor is better than theirs.

10.18 Operating systems, files, and use of memory

Operating systems

As you might guess from the preceding discussion, you frequently want to run different programs at different times, trading data back and forth between them. For instance, in writing and running a program you begin by running the editor program, creating a text file from the keyboard (good programmers never set pencil to paper, as far as we can tell). After temporarily storing the text file, you bring in the compiler program and compile the stored text file to form an assembly-language file. You store that, bring in the assembler, and produce a relocatable machine-language file from the stored assembly-language file. Finally, the linker combines the relocatable machine code with other assembled subroutines and library routines to produce the executable machine-language program, which (at last!) you run. For all these operations you need some sort of super program to juggle things around, getting programs from disk, putting them into memory, and transferring control to the relevant programs. In addition, it would be nice if each program didn't have to contain all the commands necessary to do disk reads and writes (including interrupt handling, loading of status and command registers, etc.), or, indeed, any of the other detailed data communications tasks.

These are some of the tasks of the *operating system*, a vast program that oversees the loading and running of user programs (the ones you write) and utility programs (editor, compiler, assembler, linker, debugger, etc.), as well as the handling of I/O and interrupts, and file creation and manipulation. The operating system includes a *monitor* for user interface (you tell it to run the editor, compile a program, or run a program) and many "system calls" that permit a running program to read or write a line of text from some device, find out the time of day, swap control to another program, let several multitasking "processes" share CPU time and communicate among themselves, bring in a program "overlay," etc. Good operating systems handle all the busywork of I/O handling, including "spooling" (the buffering of input or output data so that the program can run at the same time that data are being read or written to some device). When running under an operating system, a user program doesn't have to worry about interrupts; an interrupt is taken care of by the system, and it affects the running program only if it wants to take part in the handling of a particular device's interrupts. The whole business of successful "time sharing" (using one computer to handle many users at once), with the disk providing "virtual memory" for unlimited program size, is system programming at its finest.

Some popular microcomputer operating systems are MS-DOS (used on the IBM PC and its imitators), OS/2 (used on the PS/2, IBM's successor to the PC), UNIX (created at Bell Labs, widely used on VAX and 68000-based machines), MacOS, and VMS (company-supplied VAX operating system).

Files

The mass-storage medium in widest use currently is magnetic disk, either flexible ("floppy"), with contacting read/write heads, or rigid ("hard" disk, or "Winchester"), with flying heads. Typical storage capacities are in the range of 1Mbyte for floppies and 20–500Mbyte for small Winchesters. The data are organized into *files*. Text, user programs, utility programs (e.g., editor, assembler, compiler), libraries, etc., are all stored in similar ways, and all constitute files. Although the mass-storage medium is divided into physical blocks or sectors of well-defined size (512 bytes/sector is common), the files themselves may have any length. The operating system mercifully takes care of track/sector addressing, etc.; it gets the data you want, if you know the file name. There are all sorts of interesting details having to do with file organization that we don't have space to describe here. What is important is to understand that all those programs (editor, compiler, etc., as well as user source text, compiled programs, and even data) reside on some mass-storage device as named files, and the system can get them for you (read the next subsection, however, on "ramdisk"). In the normal course of its duties, the system does enormous amounts of file handling.

Recent additions to the mass-storage stable are based on consumer electronics media and provide very high density storage in small packages: (a) Optical disks of the kind used in audio CD players store nearly a gigabyte, as prerecorded "read-only" memory, as *WORM* memory ("Write Once, Read Many"), or (as with magnetic media) as fully erasable read/write memory. (b) Videotape, in both VHS and 8mm formats, lets you store a gigabyte of read/write memory on inexpensive tape; the major drawback is the long access time. Both storage systems use sophisticated error-correction schemes to overcome errors due to media blemishes, etc., which are a minor nuisance in the original audio/video applications of these media, but would be devastating for data or program storage if uncorrected.

Use of memory

Files are stored in some mass-storage device, but a program must reside in memory while being executed. A simple stand-alone program of the sort we'll talk about in the next chapter can be loaded almost anywhere in memory. But in a microcomputer with an operating system there are special areas reserved for special functions. For example, the MS-DOS operating system itself, along with its command interpreter, disk buffers, stack, etc., is usually loaded at the bottom of memory, taking care to put its interrupt vectors in the specific locations in low memory that the CPU requires, while the portion of MS-DOS that is in ROM is located high in memory, above the portion of memory reserved for video display buffers. When operating under an operating system, the allocation of memory for user programs will be handled by the system. This is particularly important to understand if you intend to use DMA; in that case you have to let the system figure out where your data buffer wound up, and use that as the starting address for the DMA block transfer.

The situation is even more complicated if programs are being swapped in and out of memory, or moved around in memory. There may be several programs in memory simultaneously, sharing "time slices" of the CPU in a multitasking mode. To add to the complexity, most microcomputers use "memory mapping," in which *physical* memory addresses (what's actually on the bus lines) are mapped to different *logical* addresses (where your program thinks it is). If that isn't enough to confuse you, consider "virtual memory," a feature of advanced microcomputers in which your program is diced up into little "pages," any of which may or may not be in memory at any instant; the program "pages" them in and out in a crazy quilt of frenzied activity.

No discussion of memory use is complete without mentioning *ramdisk*, which can be invoked even on relatively simple machines, if they have enough memory. The basic idea is to make memory look like disk, from the operating-system point of view; you then load into this ramdisk memory the programs that you need frequently. This can be handy during program development, when you need to keep using the editor, compiler, assembler, and linker. With ramdisk, things move along quickly, since no actual disk access is required. It does have the hazard that you can lose all your work if the computer crashes, since files are not automatically saved on disk. A related concept is a disk *cache*, in which an area of RAM holds the results of recent disk accesses.

Drivers

The computer world is rich with diversity - each month we see products using novel technologies in data storage (magnetic, optical), printers (laser, LED), networks, etc. Different hardware requires different controlling signals, with different timing requirements, etc. This would appear to create real programming problems, since publishing software designed for a dot-matrix printer, for example, would appear to be totally inappropriate for a laser typesetter.

The solution consists of software *drivers*, which are special programs designed to create a uniform programming interface to each particular piece of hardware. Thus, for example, the typesetting language TEX creates output in the form of *dvi* (device-independent) files; a printer driver (specific to the particular printer you are using) eats the *dvi* file and spits out the corresponding idiosyncratic printer codes to instruct the printer. TEX works with any printer, once you have the *dvi*-translating driver. The same sort of device independence goes for mass-storage devices such as disk drives, so that you can attach any of a variety of disks to UNIX, PC-type, or Macintosh computers.

Drivers are really part of the overall system software, and the average computer user is unaware of their operation. If you are designing new computer hardware, however, you will probably find yourself quickly becoming an expert on these essential software modules, since you will have to write your own drivers to make your hardware play with the rest of the team.

DATA COMMUNICATIONS CONCEPTS

A small computer system will usually be configured with some mass-storage devices, such as disks and tape, and some "hardcopy" or interactive devices, such as alphanumeric terminals, printers, plotters, etc. In addition, it may have a modem (modulator-demodulator) so that it can dial up other computers through ordinary telephone lines. Finally, local area networks (LANs) are becoming increasingly popular. With a LAN you can have access to files stored in other computers on the network, as well as the ability to share expensive resources (for example, large disks, tape drives, printers, and typesetters). In each case your CPU has to communicate data. Let's see how it works.

Incompatibility

In the dark "middle ages" of computers (say, up to 1975) the situation was pretty bleak. Each brand of computer had its own bus structure and interfacing protocol (not to mention programming language). You bought (or sometimes built) interfacing cards to fit the particular computer, with custom cables going from the interface to the peripheral itself. This general lack of compatibility extended to the peripherals themselves: You couldn't hook a tape drive to a disk interface, or a terminal to a plotter interface, etc. To make matters worse, the peripherals offered by different manufacturers generally used different signals and data-transfer conventions and were not "plug-compatible."

Compatibility

Some of this incompatibility was unavoidable, since to maximize performance different peripherals transfer their data to and from the interface differently. For example, a magnetic disk moves words in parallel byte-wide format for high speed, and the corresponding interface must use DMA transfer, as we explained earlier; by contrast, a keyboard terminal uses a standardized alphanumeric bit-serial format, with the interface using simpler interrupt-driven programmed I/O. Although some of this incompatibility is still with us, the situation is vastly improved, with most of the industry standardizing on a few agreed-upon data communications standards. The introduction of the IBM PC defined a much-needed small-machine format and data bus, while nonproprietary high-performance buses like VME and Multibus became the backplane for a number of other computers. You can get interface cards for these buses (and others, like DEC's Q-bus) from many manufacturers, which simplifies things enormously. Even more important, the manufacturers of peripherals have agreed on a few standardized "cable interfaces." The most important of these are (a) RS-232 serial format, usually used with alphanumeric ASCII data, (b) Centronics' parallel printer format, (c) SCSI parallel bus, (d) IPI bus, and (e) IEEE-488 (GPIB) instrument bus. Let's take a look at these, and then finish the chapter with a brief description of two popular kinds of local area networks, Ethernet and token-ring networks.

10.19 Serial communication and ASCII

As mentioned earlier, alphanumeric communication between a computer and devices of moderate speed is most frequently done using the 7-bit ASCII code (American Standard Code for Information Interchange), with bit-serial transmission over a single line. Table 10.3 presents a listing of the 7-bit codes. Devices communicating via serial ASCII almost always send an 8th bit, but it is not part of the ASCII code; it is most often a hardware parity bit (sometimes odd parity, sometimes even, but most often set to 0 and ignored), but it is occasionally used as a "meta" shift key to generate an additional 128 characters, which may be Greek symbols, alternate fonts, etc. There are no standards for these extra symbols. (The 8th bit also gets used when you ship *binary* data via a serial connection; this doesn't always work, though, because serial data links are so used to getting rid of the 8th bit during ASCII transfer that they may not permit you to retain it as data.)

A few notes on the ASCII table. The upper-case alphabet begins at 41_H; setting bit 5 to a 1 generates the corresponding lower-case character. The ASCII value for a digit is just the digit plus 30_H. The first 32 ASCII characters are nonprinting, or "control" characters. Some of them are important enough to have earned their own keys on keyboards, for example CR (which may be labeled "return," since keyboards don't have carriages), BS ("backspace"), HT ("tab"), and ESC ("escape"). You can generate any control character (including the above) by holding down CTRL and typing the corresponding letter from the upper-case alphabet; for example, CR is CTRL-M (try it on your computer). The control characters are used to control printing or program execution, or they can be used by programs that otherwise expect to receive alphanumeric characters, e.g., text editors. Some other important control characters, besides the ones listed above, are NUL (null), a character of all zeros often used to delimit character strings; FF (form feed), used to begin a new page; ETX (end of text, affectionately called "control C"), which many operating systems interpret as a command to abort a running program; DC3 (control S), used as a "soft handshake" to stop serial transmission; and DC1 (control Q), the complementary character to resume transmission.

Unfortunately, ASCII doesn't provide for subscripts, exponents, or any Greek or scientific characters. As a minimum, it would be nice to have π, μ, Ω, and the degree symbol (°), which crop up frequently in technical writing. Of course, it is possible to use a control character (or sequence of characters) to indicate a change of font or alphabet. This is

TABLE 10.3. ASCII CODES

non-printing					printing			printing			printing		
Name	Control char	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec
null	ctrl-@	**NUL**	00	00	**SP**	20	32	**@**	40	64	**‘**	60	96
start of heading	ctrl-A	**SOH**	01	01	**!**	21	33	**A**	41	65	**a**	61	97
start of text	ctrl-B	**STX**	02	02	**"**	22	34	**B**	42	66	**b**	62	98
end of text	ctrl-C	**ETX**	03	03	**#**	23	35	**C**	43	67	**c**	63	99
end of xmit	ctrl-D	**EOT**	04	04	**$**	24	36	**D**	44	68	**d**	64	100
enquiry	ctrl-E	**ENQ**	05	05	**%**	25	37	**E**	45	69	**e**	65	101
acknowledge	ctrl-F	**ACK**	06	06	**&**	26	38	**F**	46	70	**f**	66	102
bell	ctrl-G	**BEL**	07	07	**’**	27	39	**G**	47	71	**g**	67	103
backspace	ctrl-H	**BS**	08	08	**(**	28	40	**H**	48	72	**h**	68	104
horizontal tab	ctrl-I	**HT**	09	09	**)**	29	41	**I**	49	73	**i**	69	105
line feed	ctrl-J	**LF**	0A	10	*	2A	42	**J**	4A	74	**j**	6A	106
vertical tab	ctrl-K	**VT**	0B	11	**+**	2B	43	**K**	4B	75	**k**	6B	107
form feed	ctrl-L	**FF**	0C	12	**,**	2C	44	**L**	4C	76	**l**	6C	108
carriage return	ctrl-M	**CR**	0D	13	**-**	2D	45	**M**	4D	77	**m**	6D	109
shift out	ctrl-N	**SO**	0E	14	**.**	2E	46	**N**	4E	78	**n**	6E	110
shift in	ctrl-O	**SI**	0F	15	/	2F	47	**O**	4F	79	**o**	6F	111
data line escape	ctrl-P	**DLE**	10	16	**0**	30	48	**P**	50	80	**p**	70	112
device control 1	ctrl-Q	**DC1**	11	17	**1**	31	49	**Q**	51	81	**q**	71	113
device control 2	ctrl-R	**DC2**	12	18	**2**	32	50	**R**	52	82	**r**	72	114
device control 3	ctrl-S	**DC3**	13	19	**3**	33	51	**S**	53	83	**s**	73	115
device control 4	ctrl-T	**DC4**	14	20	**4**	34	52	**T**	54	84	**t**	74	116
neg acknowledge	ctrl-U	**NAK**	15	21	**5**	35	53	**U**	55	85	**u**	75	117
synchronous idle	ctrl-V	**SYN**	16	22	**6**	36	54	**V**	56	86	**v**	76	118
end of xmit block	ctrl-W	**ETB**	17	23	**7**	37	55	**W**	57	87	**w**	77	119
cancel	ctrl-X	**CAN**	18	24	**8**	38	56	**X**	58	88	**x**	78	120
end of medium	ctrl-Y	**EM**	19	25	**9**	39	57	**Y**	59	89	**y**	79	121
substitute	ctrl-Z	**SUB**	1A	26	**:**	3A	58	**Z**	5A	90	**z**	7A	122
escape	ctrl-[	**ESC**	1B	27	**;**	3B	59	**[**	5B	91	**{**	7B	123
file separator	ctrl-\	**FS**	1C	28	**<**	3C	60	\	5C	92	|	7C	124
group separator	ctrl-]	**GS**	1D	29	**=**	3D	61	**]**	5D	93	**}**	7D	125
record separator	ctrl-^	**RS**	1E	30	**>**	3E	62	**^**	5E	94	**~**	7E	126
unit separator	ctrl-_	**US**	1F	31	**?**	3F	63	_	5F	95	**DEL**	7F	127

the usual method used in technical word processing, where the formatter interprets subsequent ASCII characters differently. This is probably the best solution anyway, since, given the variety of symbols needed for any serious technical writing, you wouldn't be happy for long even with a very large fixed ASCII alphabet.

Note that computer keyboards are often implemented not simply as ASCII code generators, one code per keystroke; instead, recent practice is to generate unique "key down" and "key up" codes for each key. Special system software (a "keyboard driver," see Section 10.18) may then translate the keystrokes into vanilla

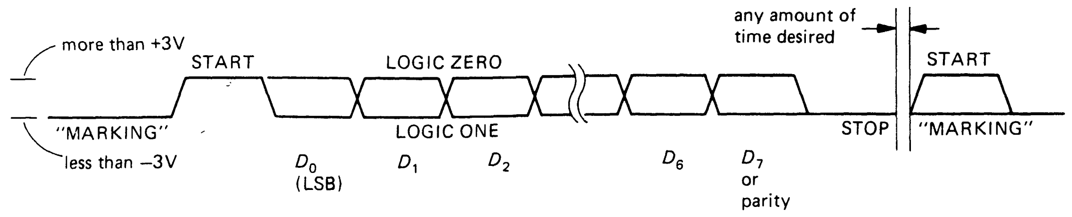


Figure 10.16. RS-232 serial data-byte timing waveform.

ASCII. However, this implementation allows much greater flexibility, since you can configure the keyboard driver to have auto-repeat keys, multiple shifts, keyboard remapping (e.g., a Dvorak keyboard), "hot keys," etc.

Bit-serial transmission

ASCII (or any other alphanumeric code) can be transmitted either as a parallel 8-bit group (8 separate wires) or as a serial string of 8 bits, one after the other. For transmission at low to moderate speeds it is most convenient to use serial transmission, to simplify wiring. A modem (discussed later in this section) converts a serial bit stream to an audio signal, and vice versa (e.g., by using one audio tone for "1," another for "0"), which can then be sent via telephone lines; serial transmission is a natural here, too. Serial transmission has a standard bit-transmission protocol and fixed bit rates: With *asynchronous* transmission, a *start* bit and a *stop* bit (sometimes two) are attached to the ends of each 8-bit character, forming a 10-bit group. The sender and receiver use a fixed bit rate, the most popular of which are 300, 1200, 2400, 4800, 9600, and 19,200 baud (= clock periods per second). Figure 10.16 shows the idea.

When no information is being sent, the transmitter sits in the "marking" state (the language comes from the teletypewriter days, with "mark" and "space"). Every character begins with a START bit, followed by the 8 ASCII bits, least significant bit first (usually organized as 7 data bits, plus 1 optional parity bit), and a final STOP bit; the latter must be at least one clock period, but may extend any amount longer. At the receiving end, a UART ("universal synchronous/asynchronous receiver/transmitter," see Section 11.11) operating at the same baud rate synchronizes to each 10-bit group, generating successive 8-bit parallel data groups from the input serial string. By resynchronizing on the START and STOP bits of each character, the receiver doesn't require a highly accurate clock; it only has to be accurate and stable enough for the transmitter and receiver to stay synchronized to a fraction of a bit period over the time of one character, i.e., an accuracy of a few percent. The receive UART is triggered by the transition at the beginning of the START bit, waits for half a bit cell to be sure the START bit is still present, and then examines the data value at the middle of each data cell. The STOP bit terminates the character and is the resting state if no new characters are sent immediately. The receive UART looks for the STOP bit level 10.5 bit-cell intervals after the START transition, to help verify a correctly sent character. "Break" is a continuous space, which cannot occur during normal character transmission. Programmable baud-rate generators (i.e., programmable dividers) are available that generate any of the standard baud rates from a single oscillator input frequency, with the output baud rate selected by a binary input code. Most modern UARTs (for example the dual-channel synchronous/asynchronous 8530 from Zilog) include internal software-programmable baud-rate generators.

RS-232

The actual serial ASCII signals can be sent in one of several ways. The original method, which dates back many decades, consists of switching a 20mA (or sometimes 60mA) current at the selected baud rate. This is known as "current-loop" signaling. It is sometimes available as an option, but has been superseded for moderate baud rates by the EIA RS-232C standard of 1969 (and subsequent RS-232D standard of 1986), which uses bipolarity *voltage* signaling. The RS-232 standard specifies the properties of both drivers and receivers: A driver must generate voltage levels of +5 to +15 volts (logic LOW input), and −5 to −15 volts (logic HIGH input), into a load of 3k to 7k, with a slew rate of less than 30V/μs, and the ability to withstand a short to any other output (which can be as inhospitable as ±5V@500mA); a receiver must present a 3k to 7k load resistance, converting an input of +3 to +25 volts to logic LOW, and an input of −3 to −25 volts to logic HIGH. Note that logic 1 gets inverted by the RS-232 driver to a negative level, called "mark"; logic 0 is a positive level ("space"). In current-loop transmission, current flows during logic 1 (mark), and ceases during logic 0 (space).

RS-232 receivers usually have voltage hysteresis at the input, and some types let you limit the response speed with a capacitor, to reduce susceptibility to noise pulses. Look at Sections 9.14 and 14.17 for a discussion of official RS-232 driver and receiver ICs. RS-232 works well up to 38,400 baud over distances of tens of feet, even with unshielded bundled multiwire cable; for short links it is sometimes used at 115,200 baud.

RS-232 also specifies the connector type and pin assignments. Unfortunately, it doesn't specify enough! This is an eternal source of confusion because, in general, two RS-232 devices, when connected together, won't work. The problem is so annoying that readers of the previous edition of this book have even complained to *us*, because we didn't tell them what to do about it. Luckily for you, you're reading the second edition. Here's the story:

There are two basic problems in this business: (a) There are two flavors of device defined, with input pins of one type corresponding to output pins of the other; you may want to connect two similar devices together, or you may want to connect two complementary types together. (b) There are five "handshaking" signals; some devices send them out, and expect to receive them back, while others ignore their inputs (and don't drive their outputs). To make things work, you've got to understand these in detail. Let's plunge in.

RS-232 was designed for connecting DTEs ("data terminal equipment") to DCEs ("data communication equipment"). A terminal always looks like a DTE, and a modem always looks like a DCE; but other devices, including microcomputers, can be either. The IBM PC looks like a DTE with a male connector, although most large computers are DCE-like. When you connect a DTE to a DCE, you just connect corresponding pins of their DB-25 connectors (which can be either male or female, at either end!), and, with some luck, it may work. We say *may*, because it still depends on which handshaking lines each device expects from the other, and bothers to drive itself. (Of course, even when the cable is right, you still have to agree on baud rate, parity, and a few other software parameters!) When you want to connect two *similar* devices, on the other hand, you can't connect corresponding pins, because that would connect the two outputs together: A DTE transmits on pin 2 and receives on pin 3, while a DCE does the reverse. So you have to connect them with a cable (called a "null modem") that criss-crosses

TABLE 10.4. RS-232 SIGNALS

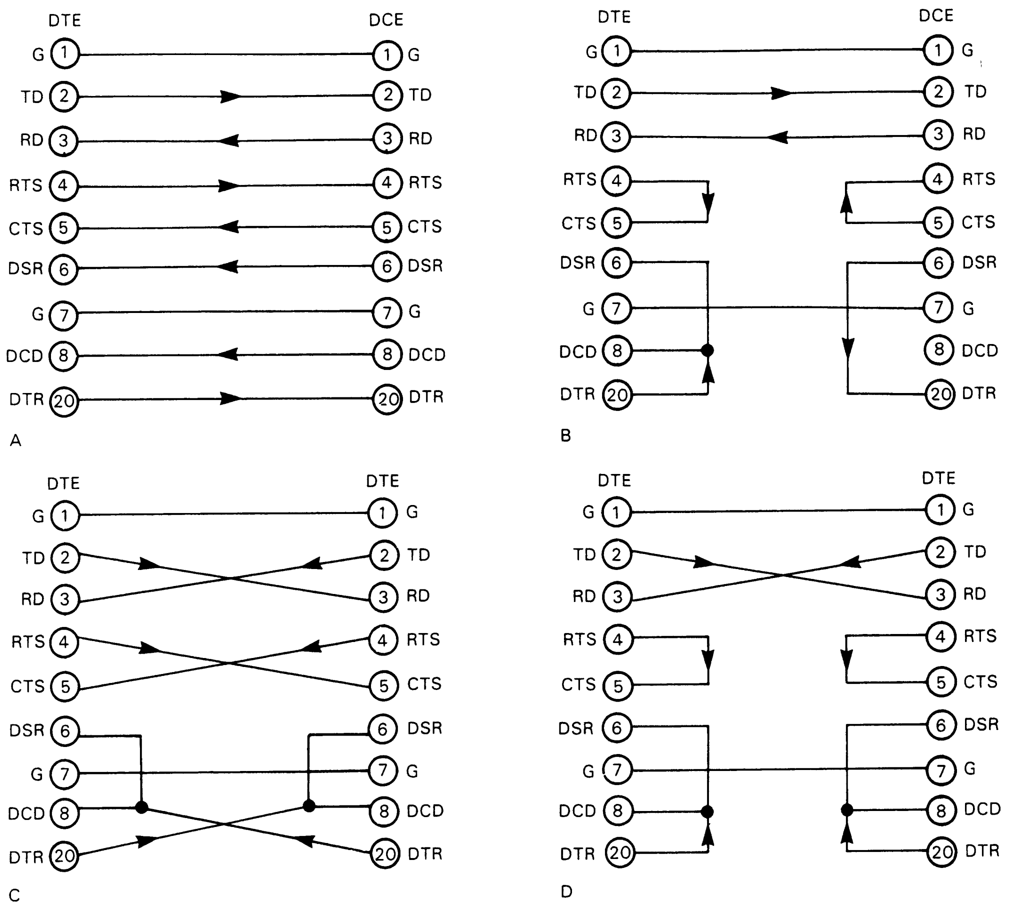
Name	Pin number 25-pin	9-pin	Direction (DTE↔DCE)	Function (as seen by DTE)	
TD	2	3	→	transmitted data	} data pair
RD	3	2	←	received data	
RTS	4	7	→	request to send (= DTE ready)	} handshake pair
CTS	5	8	←	clear to send (= DCE ready)	
DTR	20	4	→	data terminal ready	} handshake pair
DSR	6	6	←	data set ready	
DCD	8	1	←	data carrier detect	} enable DTE input
RI	22	9	←	ring indicator	
FG	1	–		frame ground (= chassis)	
SG	7	5		signal ground	

pins 2 and 3. Unfortunately, that's not all there is to it.

Table 10.4 shows all the important lines. TD and RD are the data transmit and receive lines; RTS and CTS are "ready to send" and "clear to send"; DTR, DSR, and DCD are "data terminal ready," "data set ready," and "data carrier detect." There are, in addition, two grounds: a "frame ground" (or chassis, pin 1) and a "signal ground" (pin 7); most machines just tie them together. The five signals that aren't data are *handshaking*-type control signals: A DTE asserts RTS and DTR when it's ready to receive, and a DCE asserts CTS and DSR when it is ready to receive. Some DTEs also expect their DCD input to be asserted before they will do anything. All signal lines are RS-232 bipolarity levels, with data (TD, RD) asserted *negative*, but control lines (RTS, CTS, DSR, DTR, DCD) asserted *positive.*

Note that the signal names make sense only as viewed by the DTE: For instance, pin 2 is called TD ("transmitted data") by *both* sides, even though the DTE asserts it and the DCE receives it. Thus, the name of a pin isn't enough to tell you if it's an input or output – you also need to know whether the device thinks it's a DTE or a DCE (or you can cheat and use a voltmeter!).

If all RS-232 devices asserted everything they are supposed to and listened to everything they are supposed to, then you could just connect corresponding pins (for DTE ↔ DCE), or cross corresponding pairs (for DCE ↔ DCE, or DTE ↔ DTE). However, when you connect a device that ignores all handshaking lines to one that expects them, nothing happens. So you have to tailor your strategy to the reality; this sometimes involves trickery. Figure 10.17 shows how to make cables that actually work, for all (well, *nearly* all) situations. In part A we show the connection for DTE ↔ DCE when both devices use full handshaking. RTS/CTS is one pair of handshakes, and DTR/DSR is the other. In C we show the same thing, but with a "null modem" cable to cross inputs and outputs for a DTE ↔ DTE pair. The same cable works for a DCE ↔ DCE pair, but you should reverse the arrows in the picture, and omit the connections to pin 8. These cables won't work, though, if one device is looking for handshaking and the other isn't providing it. In that case the easiest thing



25-pin		9-pin
protec. GND	1	—
TD	2	3
RD	3	2
RTS	4	7
CTS	5	8
DSR	6	6
sig. GND	7	5
DCD	8	1
DTR	20	4
RI	22	9

E

Figure 10.17. RS-232 cables that really work. Pin numbers shown are for 25-pin ("DB-25") connectors, with alternate 9-pin numbering shown in part E.

is to wire the cable so that each device provides its own handshakes, i.e., tells itself to go ahead. That's shown in B for DTE ↔ DCE and in D for DTE ↔ DTE (works also for DCE ↔ DCE, but you should omit the connections to pin 8).

How to become an RS-232 genius. If you make up these four cables, with a male *and* female connector at each end, you can make anything work with anything else (almost). Your colleagues will swear you're a genius. They will, that is, unless they've discovered the *real* professional's gimmick, an "RS-232 breakout box." It has LEDs for each line, so that you can see who is asserting what, and it has little jumpers so that you can connect any given pin to any other pin. Instructions: Look at the lights to get TD and RD connected right, then look again to see who asserts the handshakes. If a device asserts RTS, it probably looks at CTS. If both do, connect them together; otherwise, loop its RTS back to its CTS. Play the same game with DTR and DSR. If only one pair of handshakes is implemented, it is usually DTR/DSR. In general, the DTR/DSR pair is used to make sure the other side is connected and turned on, while the RTS/CTS pair is used to start and stop transmission as one side gets ahead of the other.

If you're too cheap to buy a breakout box, use a voltmeter to check for implemented signals: Any line with a large (>4V) negative or positive level is asserted; any line floating near ground is not.

Software handshaking. Some devices use the RTS/CTS hardware handshakes to start and stop data transmission while the slower device (e.g., a printer) catches up. Others transmit a "software handshake": CTRL-S (to stop) and CTRL-Q (to resume). If you're lucky, you'll have a choice. The software method means you can use a simpler cable, and if the devices ignore the the hardware lines altogether, your cable is extremely simple, with only pins 1, 2, 3, and 7 connected (all you have to figure out is whether or not to cross pins 2 and 3). The devices may still expect the hardware handshakes to be connected to enable the link, even if they use CTRL-S and CTRL-Q for detailed handshaking. In that case you can get away with the scheme of Figure 10.17B,D. Just make sure you remember to turn on the power at both ends, because neither side has any way to know that the other is alive, or even connected!

Other serial standards: RS-422, RS-423, and RS-485

The RS-232C standard was frozen in 1969, when serial data communication was a relatively leisurely occupation. It works well up to 50 feet, at speeds up to 19,200 baud. But computer and peripheral speeds have been doubling every year or two, and a better standard for serial communication was needed.

As we discussed in Section 9.14, RS-423 is an improved bipolarity single-ended protocol, good to 100kbaud and to 4000 feet (not at the same time); it is essentially compatible with RS-232. RS-422 is a unipolarity differential protocol good to 10Mbaud and to 4000 feet (see Fig. 9.37 for the speed/length tradeoff). RS-485 is similar to RS-422, but with additional specifications so that many drivers and receivers can share a single line. Table 10.5 summarizes the characteristics of these four standards.

Modems

As we remarked earlier, a *modem* ("*mo*dulator/*dem*odulator") is used to convert bit-serial digital quantities into analog signals that can be sent over telephone lines or other transmission paths (Fig. 10.18). An *internal* modem plugs into a slot in your computer (or comes built-in), whereas an *external* modem is a stand-alone box, powered from the ac power line, with RS-232 connection to

TABLE 10.5. SERIAL DATA STANDARDS

	RS-232C/D	RS-423A	RS-422A	RS-485
Mode	single-ended	single-ended	differential	differential
Maximum number				
drivers	1	1	1	32
receivers	1	10	10	32
Maximum cable length	15m	1200m	1200m	1200m
Maximum data rate (bits/s)	20k	100k	10M	10M
Transmit levels	±5V min ±15V max	±3.6V min ±6.0V max	±2V min (diff'l)	±1.5V min
Receive sensitivity	±3V	±0.2V	±0.2V	±0.2V
Load impedance	3k to 7k	450Ω min	100Ω min	60Ω min
Output current limit	500mA to V_{CC} or gnd	150mA to gnd	150mA to gnd	150mA to gnd 250mA to -8V or +12V
Driver Z_{out}, min (pwr off)	300Ω	60k	60k	120k

your computer's serial port. In either case the modem communicates with the telephone line, in one of two ways: (a) direct connection, via a telephone-type "modular jack," or (b) "acoustically coupled," by seating the telephone handset into a rubbery cradle containing microphone and speaker. Acoustically coupled modems are pretty much out of style these days, although they can be handy in hotel rooms where you may not want to crawl around under the beds looking for a modular jack (which may not even exist!).

In most situations you want to be able to send data on a single telephone channel in both directions simultaneously ("full duplex"), sharing the telephone audio bandwidth, which is roughly 300Hz–3kHz. There are three full-duplex formats in common use: 300 baud FSK (Bell 103), 1200 baud dibit PSK (Bell 212A), and 2400 baud dibit PSK (FSK stands for "frequency-shift keying," and PSK stands for "phase-shift keying"). A modem designed for 1200 baud, say, generally also supports 300 baud communication, etc. Although you don't need to understand how the modem encodes its data in order

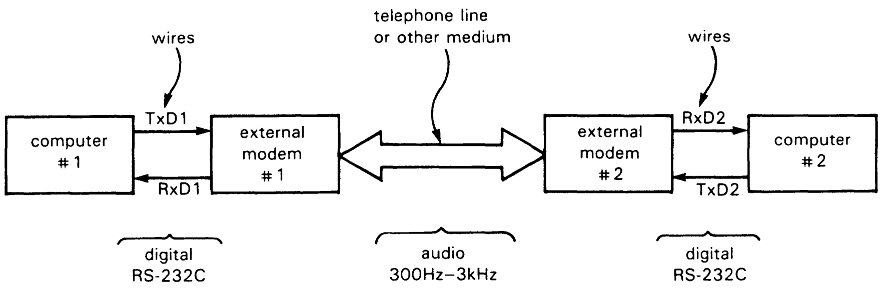
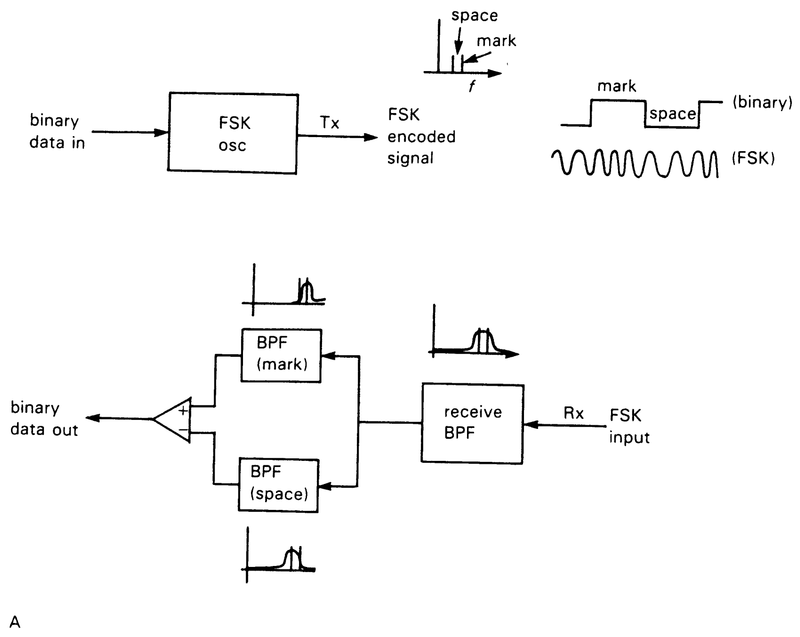


Figure 10.18. Modem communication.



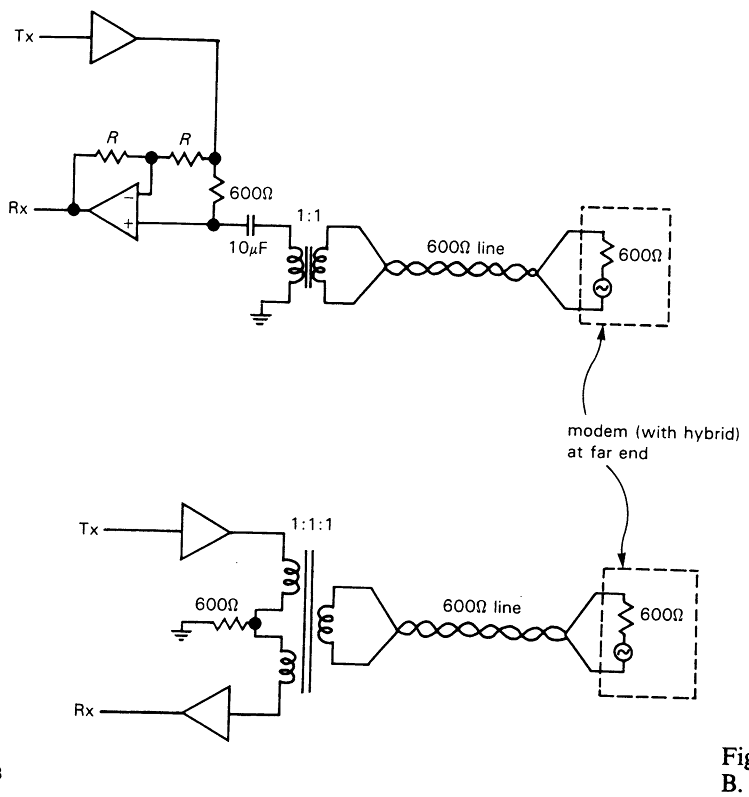


Figure 10.19. A. FSK modem. B. Hybrid couplers

to use it, the methods are interesting in their own right, and we can't resist describing them briefly.

The 300 baud standard (Bell 103) uses frequency-shift keying (FSK), in which a designated pair of audio tones represents mark and space: 1270Hz (mark) and 1070Hz (space) in one direction, 2225Hz and 2025Hz in the other. A Bell 103 modem is very simple, with a switchable oscillator for transmitting, and a pair of audio filters for receiving (Fig. 10.19A). Note also the use of a *hybrid* circuit (Fig. 10.19B) to isolate the outgoing signal from the received signal: Assuming the telephone line is close to its nominal 600 ohms impedance, none of the modem's own transmitted signal (Tx) appears back at its received-signal (Rx) output. In practice, hybrids don't work that well, because the telephone line impedance can deviate substantially from the nominal 600 ohms (see Section 14.5). Thus, it is important to have a very sharp receive filter, which adds some complexity to the modem circuit.

EXERCISE 10.5

Figure out how the hybrid circuits in Figure 10.19 work. Then impress your friends with your new knowledge.

The 1200 baud standard (Bell 212A) works differently. The digital data stream is grouped into bit *pairs* ("dibits"); each of the four possible dibits is transmitted as a designated phase shift of a fixed-frequency carrier (00: +90°, 01: 0°, 10: 180°, and 11: −90°), with smooth transitions of phase from each transmitted dibit to the next. Thus, dibits are transmitted at a 600Hz rate. The (phase-modulated) carrier frequency is 1200Hz in one direction, 2400Hz in the other. The receiving modem decodes by looking at the *difference* in phase of adjacent dibits. This clever idea has one pitfall, namely that the receiver loses track of relative phase if there is a long run of similar dibits. Therefore, in order to prevent long runs of constant phase, the transmitted data stream is randomized by exclusive-ORing it with a pseudo-random sequence (generated by a 17-bit shift register with XOR feedback from the 14th bit, see Section 9.32), with an identical descrambling process at the receiving end.

The 2400 baud full-duplex modems also use phase-encoded dibits, though with a different set of phases. These sophisticated modems tend to use real-time adaptive equalizers to correct the frequency and time-delay errors of the telephone line, and highly optimized filters for both transmitted and received signals. The end result is that the error rate is not significantly degraded when compared with the earlier 300 baud FSK modems.

You don't have to construct a modem from scratch, because complete modem chips and modules are made by AMI/Gould, Exar, National, Rockwell, Silicon Systems, and TI. Your life is made even easier, however, if you buy a complete modem, whether in the form of an internal plug-in card or an external box with RS-232 connection to your computer. Modems cost \$100–\$300, depending on features. Look for "Hayes-compatible" modems, which accept standardized commands for dialing, etc., that are now the de facto standard used by all communications software.

Some good advice: When using a modem to transfer data files between computers, be sure to use a block-checking modem protocol such as Kermit or XMODEM. These send the data in fixed-length blocks, each with error-checking checksums. The receiving modem compares the checksums, automatically insisting on retransmission of bad blocks. Files received this way are guaranteed error-free; files sent with plain unformatted ASCII transmission, by contrast, can almost be guaranteed to have errors!

10.20 Parallel communication: Centronics, SCSI, IPI, GPIB (488)

For cable communications with high-speed peripherals, parallel transmission is generally better than serial. Here are the popular favorites.

Centronics

This is a simple byte-wide unidirectional parallel port with handshaking, originated by Centronics and now widely used for printers. Unlike RS-232, *it always works!* Table 10.6 lists the signals, which are supposed to be sent with twisted-pairs and terminated in a 36-pin connector. Figure 10.20 shows the corresponding timing.

The basic signals are listed in the first group: D0–D7, STROBE′, ACKNLG′, and BUSY. BUSY is a flag: When LOW, the printer is not "busy," i.e., it's ready to accept data; the data source (computer) therefore asserts DATA, then a STROBE′ (with data guaranteed valid on both sides).

TABLE 10.6. CENTRONICS (PRINTER) SIGNALS

	Pin number			
Name	sig	com	**Direction**	**Description**
STROBE'	1	19	OUT	data strobe
D0	2	20	OUT	data LSB
D1	3	21	OUT	•
D2	4	22	OUT	•
D3	5	23	OUT	•
D4	6	24	OUT	•
D5	7	25	OUT	•
D6	8	26	OUT	•
D7	9	27	OUT	data MSB
ACKNLG'	10	28	IN	finished with last char; pulse
BUSY	11	29	IN	not ready (note 1)
PE'	12	30	IN	HIGH = no paper
SLCT	13	–	IN	pulled HIGH
AUTO FEED XT'	14	–	OUT	auto LF
INIT'	31	16	OUT	initialize printer
ERROR'	32	–	IN	can't print (note 2)
SLCT IN'	36	–	OUT	deselect protocol (note 3)
GND	–	33	–	additional ground
CHASSIS GND	17	–	–	chassis ground

note 1: BUSY = HIGH
i) during each char transfer
ii) if buffer full
iii) if off-line
iv) if error state

note 2: ERROR' = LOW
i) if out-of-paper
ii) if off-line
iii) if error state

note 3: normally LOW
i) sending DC3 when SLCT IN' = HIGH deselects printer
ii) can only re-select by sending DC1 when SLCT IN' = HIGH

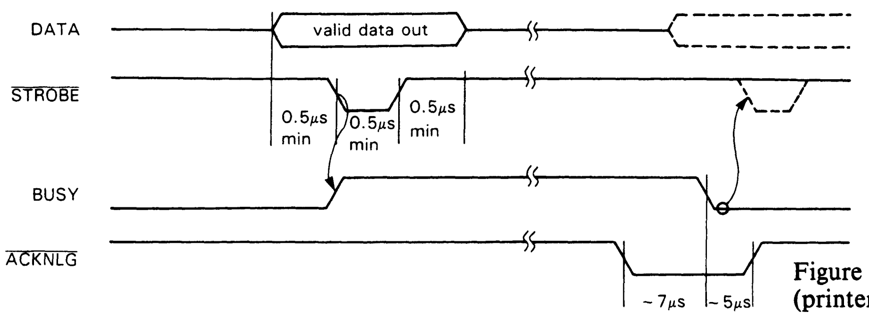


Figure 10.20. Centronics (printer) interface timing.

BUSY then goes HIGH, and it comes LOW again only when the printer is ready for another byte. The computer should look at BUSY, as shown, in order to know when it can send another byte. ACKNLG′ (which is a pulse, not a level) can be used to trigger an interrupt; don't try to use it instead of BUSY, though, because it may be gone by the time you look, and you'll wait forever.

There are several other signals, to indicate that the printer is out of paper (PE′), or off-line (ERROR′ or BUSY); the computer can initialize the printer (INIT′), ask for automatic line feed (AUTO FEED XT′), or send a byte to deselect the printer (set SLCT IN′ HIGH, then send an ASCII DC3). Note the relaxed timings, obviously intended for a slow (mechanical) device that can't accept data at a high rate. Most printers have some buffer memory, so they can accept data at a high rate initially; on the average, though, you can send bytes only at the printing rate. For a dot-matrix printer you're talking 100–300 bytes per second.

If you need to design a Centronics interface to go on some computer's bus, the easiest thing is to drive all the output lines from latched data via programmed I/O: Make D0–D7 one port, and the remaining lines (including STROBE′) a second port. For the input signals (BUSY, etc.), don't latch anything, just enable them onto the bus for programmed IN. A nice touch is to use ACKNLG′ to make an interrupt. Figure 10.21 shows the idea, for the IBM PC bus. Note that interrupts are easy here, because the PC uses edge triggering; just use the trailing edge of ACKNLG′, as shown. We've used one of the latched output bits to disable the interrupt line, as discussed in Sections 10.09 and 10.11. Note also the use of the bus signal RESET DRV to disassert all outputs (and also interrupts) at power-on; that's why we chose the '273 octal D register (which has a RESET′ input).

To use this interface, you assert and disassert output control lines selectively by sending OUT bytes to port B, with appropriate bits set to 1 or 0. With a latched output arrangement like this you can always safely change the state of one output bit without introducing glitches on the unchanged outputs. For this purpose, keep a copy in memory of the current byte latched in port B, so you can send out a new byte to port B with only one bit changed (by using AND and OR, see example below). To generate a STROBE′ pulse you must use software, since the interface has no ugly monostables. Program 10.6 shows how you make a "software pulse" on the STROBE′ line. Note the use of AND and OR, to clear and set a single bit, respectively. In this example we didn't bother updating the byte stored in "current," because at the end it was unchanged. If instead we had changed (and left changed) one of the other control bits, we would have saved the new byte with a "`MOV current,AL`" instruction at the end.

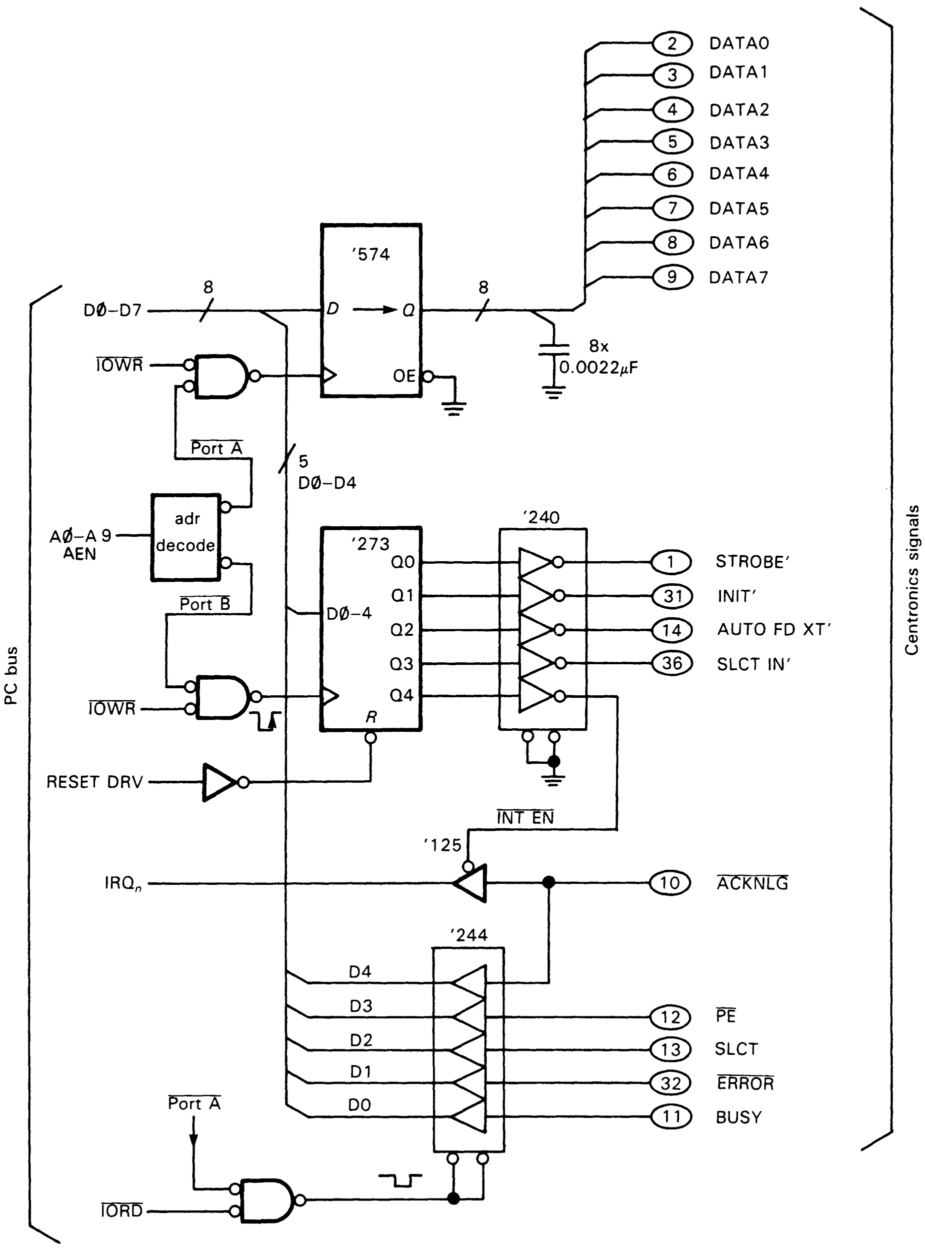


Figure 10.21. Centronics port for PC.

A hardware alternative to keeping a copy of the port byte in memory is to add a "readback" port to the interface, so a programmed IN lets you look at what's actually latched. The next example lets you discover how.

EXERCISE 10.6
Imagine that you are feeling energetic and want to add a readback port to the Centronics interface circuit. Make an IN from port B do the job. You should be pleasantly surprised at how little hardware is required.

Program 10.6

```
                                ;make a software pulse
                                ;assume Cenronics "port B" address is in DX
                                ;assume strobe bit (bit D0) is initially "1"
urrent DB 0                     ;copy of port B kept here
       o
       o
       MOV   AL,current         ;current value of control byte
       AND   AL,0FEH            ;clear D0
       OUT   DX,AL              ;send to port B
       OR    AL,1               ;set D0
       OUT   DX,AL              ;and send it out again
       o
       o
```

EXERCISE 10.7
Now rewrite Program 10.6, using your new port and omitting the use of "current."

Centronics ports are standard on nearly all microcomputers; don't hesitate to take advantage of it, if you need a quick and simple parallel output port. In many cases (but not on the IBM PC) the microcomputer will even let you use the port bidirectionally; the usual way that's done is by sending a control bit to the port to reverse the direction of the single 8-bit data path.

SCSI and IPI

These are universal parallel interface standards for connecting disks and other high-performance peripherals to microcomputers, as mentioned briefly in Section 10.16. SCSI ("Small Computer System Interface") is an 8-bit parallel cable interface with handshakes and protocols for handling multiple hosts and multiple peripherals. It has both asynchronous and synchronous modes, and defined software protocols. You can get SCSI interface cards to plug into most popular microcomputer buses, including VME and Multibus I and II; you then connect this SCSI "host adapter" to the peripheral's controller card via a flat-cable SCSI bus (Fig. 10.22). The controller card is often part of the peripheral itself (e.g., it may be attached to a hard-disk drive) and communicates with the drive by a "device-level interface," which will have a name like "ST-506/412," ESDI, or SMD.

SCSI has the advantage of effectively making all microcomputers compatible with all peripherals. Everyone's rushing to adopt SCSI, and new microcomputer designs incorporate it right on the CPU motherboard. At the peripheral end, manufacturers are eliminating the controller by going to an "embedded-SCSI" architecture, in which the SCSI bus becomes also the device-level interface. In other

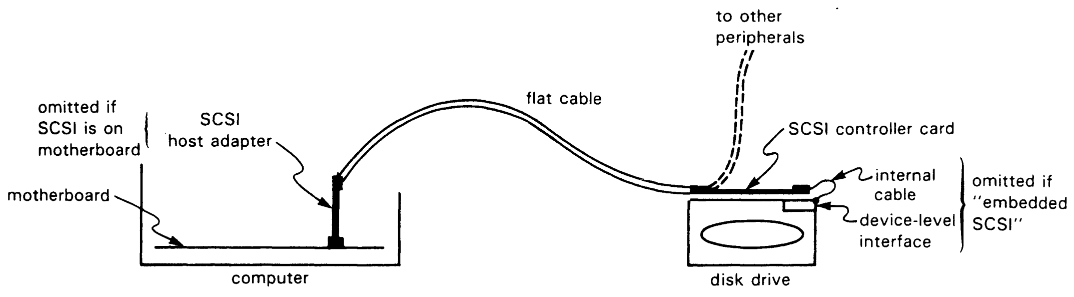


Figure 10.22. SCSI bus with single peripheral.

words, you just hook a cable from the microcomputer's motherboard to the disk drive. SCSI supports data rates to 1.5Mbyte/s (asynchronous) or 4Mbyte/s (synchronous), with cable lengths to 20 feet (single-ended) or 80 feet (differential).

SCSI is complicated enough that we don't have room here to define all its signals, modes, command protocols, and interfacing possibilities. However, because of its popularity, there are single-IC interface chips (e.g., the NCR 5380 series, Western Digital 33C90 series, and others from Fujitsu, Ferranti, etc.) to make your life easy.

SCSI works well with current-generation disks. However, in order to increase data-transfer rates, the industry is considering going to a 16-bit wide interface bus. For this the IPI ("Intelligent Peripheral Interface") may be the next interface bus of choice. IPI specifies a 16-bit parallel bus operating to 10Mbyte/s (5MHz transfer rate); like SCSI, it also works with multiple hosts and peripherals. Hard-disk drives have been getting denser and faster at an amazing pace lately; given the increasing transfer rates, the world is heading rapidly toward universal embedded-bus interfaces (SCSI or IPI). In a few years you probably won't see any other formats.

□ *IEEE-488 (GPIB, HPIB)*

When laboratory instruments first became available with actual data outputs on the back, it was a case of "each company for itself." There were nearly as many interface protocols as there were instruments, with parallel and serial modes, positive and negative polarities, and all sorts of crazy handshakes. It was total pandemonium. We remember vividly designing a huge-digit (6 inches high) display for use in Harvard's lecture halls: It had separate input circuitry for each instrument we owned!

Hewlett-Packard decided in the mid-1960s to end this craziness by defining a universal instrument interface. They modestly called it the Hewlett-Packard Interface Bus (HPIB) and implemented it as the only option on all new designs. It permits up to 15 instruments on a single bus cable up to 20 meters long, with a cleverly designed connector that you can stack at each node. The HPIB bus protocol is byte-wide with handshakes, and it allows data-transfer rates to 1Mbyte/s; it includes software commands to enable any connected device to become a "talker" (source of data), and any combination of the remaining devices to be "listeners" (recipients of data). A "controller" (dictator) tells everybody what to do.

HPIB worked so well that a standards committee was set up by the IEEE to make it official. The resulting standard is known as IEEE-488-1975/ANSI MC1.1, which everyone except HP refers to as "GPIB" ("general-purpose interface bus") or "488-bus." It has become the universal digital interface for laboratory instrumentation. The instruments of all companies can be strung together on the same GPIB, with a microcomputer (or fancy desk calculator) giving the orders. For example, you can set the waveform, frequency, and amplitude of a frequency synthesizer, then take voltage measurements from the same experiment or process.

10.21 Local area networks

In prehistoric times, computing was done in "batch mode" on large centralized computers. They were powerful (slower than the least powerful of today's personal computers, with a tiny fraction of the memory) and expensive (comparable to today's supercomputers). You punched your programs on decks of cards, then submitted the job. With luck, your aborted output was available by the end of the day, so you could resubmit the job the next morning to find the next bug.

Nowadays we're all spoiled by incredible desktop horsepower, fast disks, beautiful graphics. We want more. We want to be able to exchange files with the guy down the hall without getting out of our chairs. We want instant access to everyone's data bases, printers, and fancy peripherals. The way we get it is with networking – both the worldwide networks like BITNET and DECNET and "local area networks" (LANs) like Ethernet and LocalTalk.

The field of networking is still in its infancy, and we expect dramatic changes in the next decade. A few trends have emerged, however, and it's worth describing the kinds of LANs in use today.

CSMA/CD (Ethernet)

Ethernet typifies "carrier-sense multiple-access/collision-detection" (CSMA/CD) networks. It uses coaxial line to transmit 10Mbit/s signals to the addressed recipient. An Ethernet message is sent in "packets," with a preamble and error-checking. The sending protocol goes like this: (a) wait until you see no activity on the network; (b) begin sending your message packet (see below); (c) while sending, check simultaneously for interference (a "collision"); (d) (i) as long as all is clear, continue sending your message, but (ii) if you detect interference, jam the network intentionally (to ensure that everyone else sees the collision!), then abort your transmission, wait a random length of time, and try again; wait a longer "random" time after each successive failure.

Ethernet messages are organized into relatively short packets ($\approx$1kbyte maximum), each of which includes a *header* (identifying recipient and sender), a few bytes telling the packet's length, type, and sequence number, the actual group of data bytes, and finally a "cyclical redundancy checksum" (CRC), from which the recipient can verify error-free transmission. Note that a collision can occur only during the beginning of transmission of a packet, since [by rule (a) above] a transmission in progress for twice the network travel time will not be interfered with.

Ethernet was invented by Xerox and is widely used. It has ample bandwidth for most local area networks, and its performance degrades somewhat gracefully under heavy use, owing to the random retry protocol. You can get Ethernet controllers for most serious microcomputers (VAX, IBM PC, etc.) and buses (Multibus, VME), and it's the official network for the popular Sun and NeXT workstations. An Ethernet network can go up to 1km per segment, with up to 2 repeaters; you can also have fiber-optic "bridges" of greater length. A number of desktop computers can share a multiple-port RS-232 "server," tied into one node on the Ethernet coax. Servers can also tie into shared resources such as printers and large disks.

Token-ring networks

A token-ring network visits a closed set of nodes, in a ring configuration. Collisions are not allowed here, and the rules of the game go like this: Imagine some token object; whoever has it is permitted to send messages, while all others can only listen. In a token ring, the token is a short message that can be passed around when the owner is finished. At any time, one node owns the token and is free to send messages. As with Ethernet (and any other sensible network), the messages are packetized, often using the SDLC format ("Synchronous Data Link Control": one packet = flag + address + header + message + checksum + flag). The message packets circulate around the ring until the addressed recipient receives them. When the sender is finished sending the full message (normally many packets), he sends the token. It circulates around until some other node in the ring, desiring to send a message, swallows it, becoming the new token owner.

LocalTalk

LocalTalk (formerly Appletalk) is a simplified collision network, designed by (guess who) Apple Computer. It is a linear network, not a ring. One node can transmit, while all listen. The cable is a single differential pair, with RS-422 signals transformer-coupled at each node. The packet format is SDLC. Maximum network length is 1000 feet, with up to 32 nodes attached. The network bandwidth is 230.4kbit/s. A compatible variant known as PhoneNET (Farallon Computing Inc.) uses standard telephone cable and connectors and claims to work up to 4000 feet.

The protocol is similar to Ethernet, but simpler: If you hear no activity, you may send a packet. The network hardware doesn't attempt to detect collisions; it just forwards received packets with valid checksums up to the next higher level of software. A collision generally clobbers the colliding packets, rendering both their checksums invalid; thus, the software never gets the message at all! It is the software's job to notice this: For example, the sender of a message expects a reply; if he doesn't get one after a while, he initiates an identical message and tries again. LocalTalk is a "CSMA/CA" network; the "CA" stands for collision *avoidance,* rather than Ethernet's collision *detection.*

LocalTalk has defined protocols for sharing of files and resources (printers, modems, etc.), and it has a method for naming devices connected to the network. You can even get LocalTalk interfaces for non-Apple computers, letting you ship files between Macintoshes, IBM-compatibles, and UNIX computers, and to shared resources such as laser printers.

□ 10.22 Interface example: hardware data packing

If all your instruments connect to a standardized interface bus (such as the GPIB), you're in great shape: Just buy the interface card for your computer, buy some cables, string things together, and hire a programmer. It doesn't take much talent, only money. However, this chapter is about bus interfacing, so we would like to conclude with a complete design example.

If you're like us, you probably don't throw out all your functioning instruments when something new comes along. Some extremely capable measurement instruments were made before the era of GPIB; you can bring new life to them by cooking up an interface to your lab computer. As an example, an 8-digit frequency counter with multiplexed display is likely to have a rear-panel output that gives you one digit after another ("digit-serial, bit-parallel"), encoded as 4-bit BCD, and probably presented at the display's internal refresh rate. You have no control over the timing; each valid digit, along with its 3-bit digit-position address, is signaled with a strobe. Such an instrument most likely uses TTL output levels.

Figure 10.23 shows how to interface such an instrument to an IBM PC. This is a complete interface, including a status flag, interrupt, and selectable I/O port address. The action begins at the lower left, where the counter is busy putting out successive digits, their addresses (0–7), and a STROBE′ pulse when the data is valid. The counter goes from the least significant digit (LSD) to the most significant digit (MSD), so a complete output cycle ends with the receipt of the MSD (digit 7). The eight '173 registers (4-bit D registers with three-state outputs) latch the successive digits, being driven in parallel and separately clocked via the decoded digit addresses. Note the use of a '138 strobed 1-of-8 decoder to generate

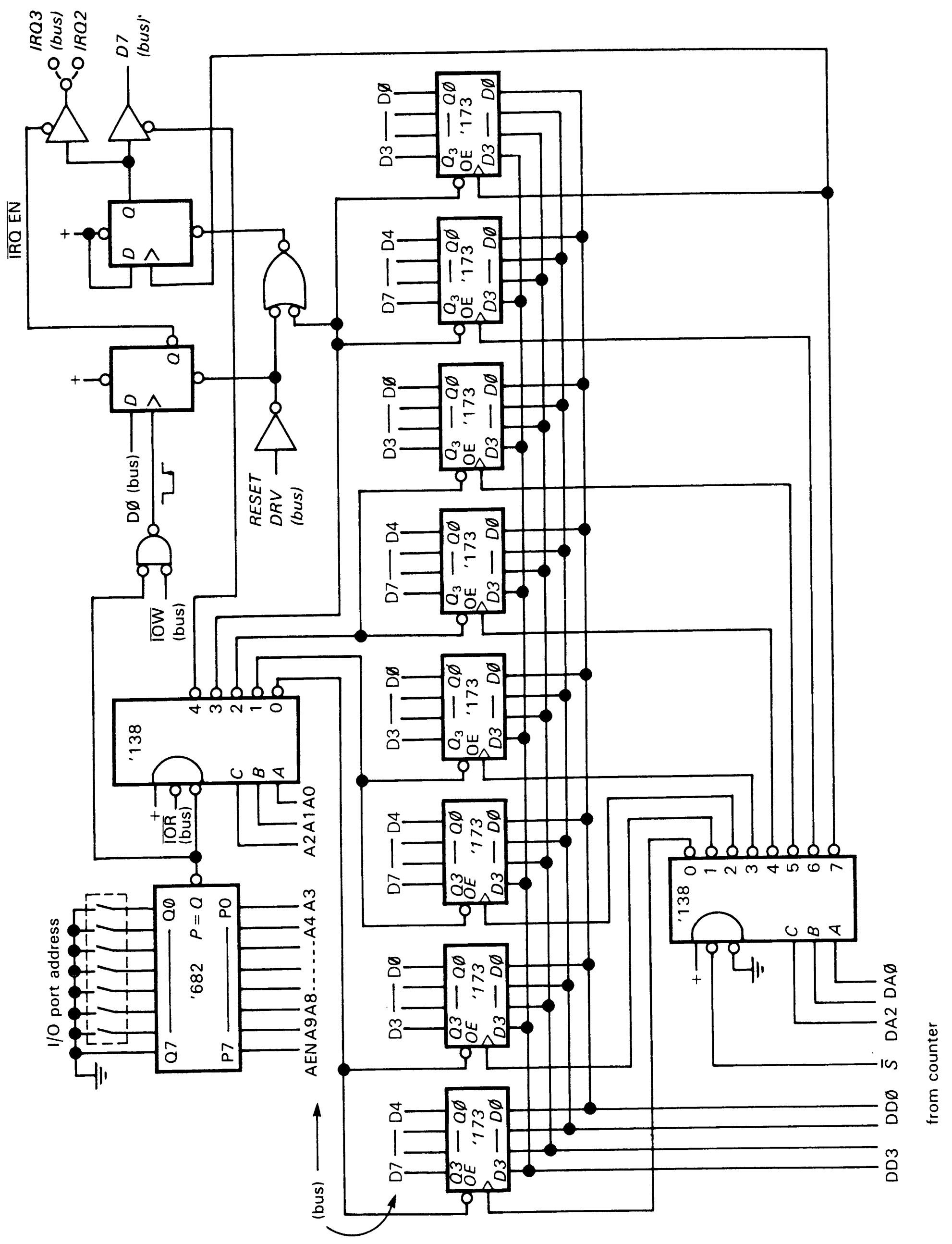


Figure 10.23. Character-serial interface.

the digit clocking signals from the address and strobe.

The counter output is thus latched in the eight 4-bit registers, with the outputs connected as four 2-digit groups (8 bits each). The PC can thus bring in all eight digits with four byte-wide data `IN` commands, from four successive I/O port addresses (beginning with the one set on the DIP switch). In fact, it can do even better by reading from a 16-bit register (i.e., doing an "`IN AX,DX`," rather than an "`IN AL,DX`"), which causes two successive byte reads from consecutive I/O port addresses.

Note the simple address decoding scheme: A '682 octal comparator generates a LOW output when the 7 high-order address bits match the switch settings (and also the nuisance AEN is LOW); this "base address" enables a '138 1-of-8 decoder, strobed by IOR′, which decodes the low-order three address bits to generate the separate data `IN` enabling pulses corresponding to successive port addresses. This is a common method of handling address decoding, since you usually assign a few contiguous port addresses to the various registers of a single interface.

The status flag is set when the last digit of each group is received from the counter; it can be read with a data `IN` from PORT+4, where PORT is the address set with the DIP switch. The flag is cleared when the CPU reads the last (most significant) data byte (from PORT+3). This interface also has provision to make interrupts, jumper-selectable on either IRQ2 or IRQ3, and enabled by sending a 1 to PORT (and disabled by sending a 0); note the lazy address decoding we've used for `OUT`, to save a gate. In a spirit of good citizenship, both the status flag and interrupt enable flip-flops are cleared at power-on.

This interface is an example of "packing" data, the process by which several numbers are stuffed into one computer word. If the "numbers" happen to consist of single bits, you can pack 16 of them into each 16-bit word. This isn't as crazy as it sounds: In digital signal processing you sometimes deal with periodically sampled "hard-clipped" waveforms (which you can think of as 1-bit A/D conversion); for highest I/O throughput rate you pack in hardware (as we did in this example) and read in bus-wide words. Of course, if speed is not important, the simplest thing is to bring in the data with the least hardware and then do the packing and conversion in software. In the preceding example, for instance, you might latch and transfer to the CPU one digit at a time if you can be sure that the latency time of the computer is short enough that no digits will be lost.

EXERCISE 10.8

Modify the interface circuit so that the IRQ line used by the interface is *programmable*: Sending 01_H to PORT enables interrupts on IRQ2, and sending 02_H to PORT enables interrupts on IRQ3; both are disabled by sending 0 to PORT, and also at power-on.

A practical note about this interface circuit: In general it is best to avoid loading bus lines excessively. Our circuit ties each Dn line to the outputs of four '173 three-state registers, which is an undesirably large capacitive load. Although our circuit would undoubtedly work properly, it might limit the number of additional cards you could plug into the bus (particularly if the others sinned in the same way!). In this example, a single '244 three-state octal buffer, interposed between the D0–D7 outputs and the PC data bus, would be a good solution. It should be enabled with the AND of the decoded port address and IOR.

10.23 Number formats

In the preceding example, the bytes (or words) brought in are not in the computer's internal binary-number format; they're really BCD, packed two digits per byte (or four per word). To do meaningful computation, it is best to convert them into an integer or a floating-point number (although there are "decimal-adjust" operations that let you do arithmetic directly on packed BCD numbers). Let's take a look at the usual number formats used in computers (Fig. 10.24), a subject we touched on briefly at the beginning of Chapter 8.

Integers

Signed integers are always represented in 2's complement, using either 1, 2, or 4 bytes, as shown. Thus, the most significant bit (MSB) tells the sign, even though 2's complement is not the same as sign/magnitude representation (e.g., -1 is 11111111, not 10000001; see Section 8.03). You can think of 2's complement as offset binary with inverted MSB; alternatively, you can think of it as an integer with the bit values as shown in the figure. Many computers let you declare variables as *unsigned* integers, in addition to 2's complement signed integers. A 2-byte unsigned integer can have values from 0 to 65535.

Floating-point numbers

Floating-point numbers, also called *real* numbers, are usually 32-bit ("single precision") or 64-bit ("double precision"), with an additional 80-bit format sometimes used for temporary values during calculations. Unfortunately there are several common representations in use. The most popular is the recently completed IEEE standard (officially known as ANSI/IEEE Std 754-1985), which has been implemented by nearly all floating-point chip sets (including Intel's 8087/287/387, Motorola's 68881, and chip sets from AMD, Weitek, et al.) and is therefore universal in microcomputers that accept those chips (this includes the IBM PC).

Figure 10.24 shows the IEEE 32-bit and 64-bit formats. The 32-bit single-precision format has 1 sign bit, 8 exponent bits, and 23 bits of fraction. The exponent tells the power of 2 that the fraction (see below) should be multiplied by. The exponent is "biased" by adding 127, so that the exponent field 01111111 corresponds to an exponent of 0; exponents thus go from -127 to $+128$. The fraction itself uses an interesting trick, originated by DEC in their floating-point format. A floating-point number in binary can always be written in the form $\text{f.fff}\times 2^e$, where f.fff is the (base-2) mantissa ("significand"), and e is the (power-of-2) exponent. In order to maximize the precision you get with a given number of mantissa bits, you "normalize" it by shifting the mantissa left (and decrementing the exponent) until the leading bit is non-zero, thus casting it in the form $1.\text{fff}\times 2^e$. Now, here's the "hidden-bit" trick: Since the resulting normalized significand always has a nonzero MSB, it would be redundant to display it; i.e., you don't put 1fff in the number, just the fff, with the leading 1 assumed. The resulting number gains one bit of precision, and has a range of $\pm 1.2\times 10^{-38}$ to $\pm 3.4\times 10^{38}$.

EXERCISE 10.9
Show that the range of normalized floating-point numbers is as claimed, by constructing the smallest and largest numbers.

The IEEE double-precision format is similar, but with the significand precision more than doubled (by attaching 29 more bits) and with the exponent fortified by an additional 3 bits. The range of numbers is as shown in the figure. There is also a whopping "extended-precision" (80-bit) format, as shown. The IEEE format allows non-normalized numbers also, to give some additional range at the small end (at the expense of precision); these "denormalized" numbers go down to $\pm 1.4\times 10^{-45}$. The standard also defines zero (e = fff = 0; thus there are two

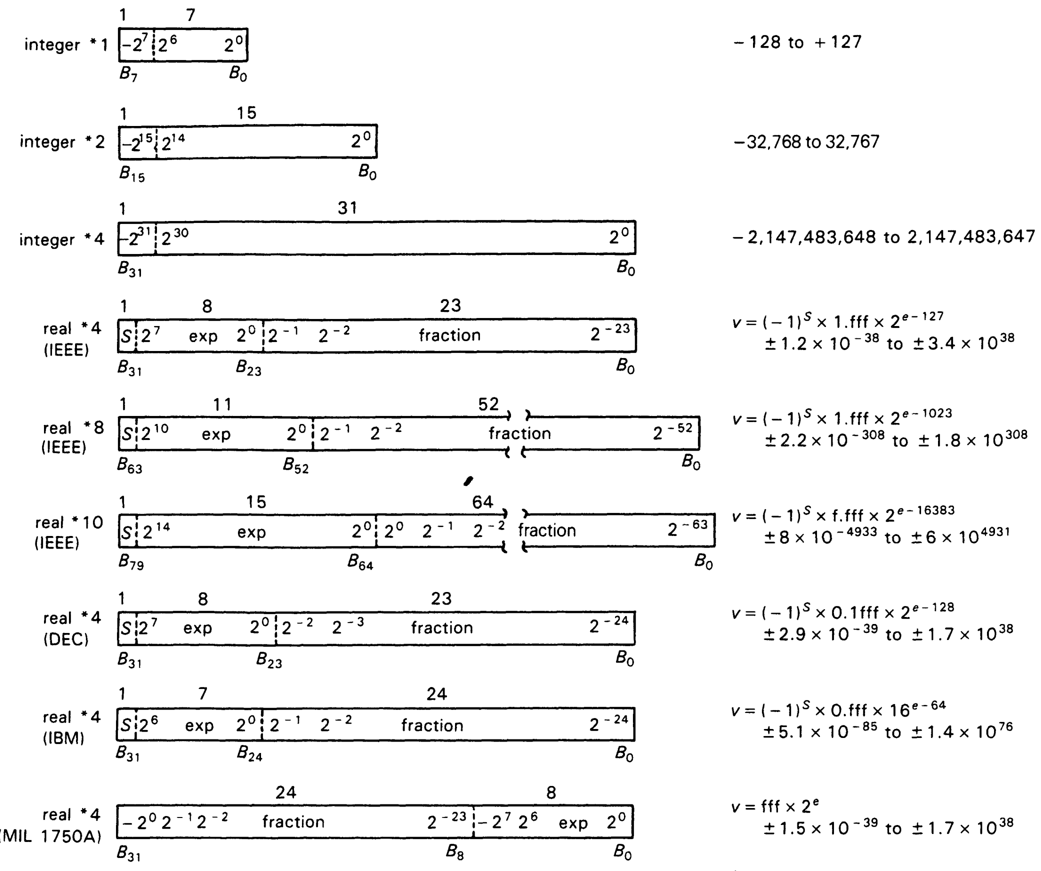

integer *1
1 7
-2^7 2^6 2^0
B_7 B_0
−128 to +127
integer *2
1 15
-2^{15} 2^{14} 2^0
B_{15} B_0
−32,768 to 32,767
integer *4
1 31
-2^{31} 2^{30} 2^0
B_{31} B_0
−2,147,483,648 to 2,147,483,647
real *4 (IEEE)
1 8 23
S 2^7 exp 2^0 2^{-1} 2^{-2} fraction 2^{-23}
B_{31} B_{23} B_0
$v = (-1)^S \times 1.\text{fff} \times 2^{e-127}$
$\pm 1.2 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$
real *8 (IEEE)
1 11 52
S 2^{10} exp 2^0 2^{-1} 2^{-2} fraction 2^{-52}
B_{63} B_{52} B_0
$v = (-1)^S \times 1.\text{fff} \times 2^{e-1023}$
$\pm 2.2 \times 10^{-308}$ to $\pm 1.8 \times 10^{308}$
real *10 (IEEE)
1 15 64
S 2^{14} exp 2^0 2^0 2^{-1} 2^{-2} fraction 2^{-63}
B_{79} B_{64} B_0
$v = (-1)^S \times \text{f.fff} \times 2^{e-16383}$
$\pm 8 \times 10^{-4933}$ to $\pm 6 \times 10^{4931}$
real *4 (DEC)
1 8 23
S 2^7 exp 2^0 2^{-2} 2^{-3} fraction 2^{-24}
B_{31} B_{23} B_0
$v = (-1)^S \times 0.1\text{fff} \times 2^{e-128}$
$\pm 2.9 \times 10^{-39}$ to $\pm 1.7 \times 10^{38}$
real *4 (IBM)
1 7 24
S 2^6 exp 2^0 2^{-1} 2^{-2} fraction 2^{-24}
B_{31} B_{24} B_0
$v = (-1)^S \times 0.\text{fff} \times 16^{e-64}$
$\pm 5.1 \times 10^{-85}$ to $\pm 1.4 \times 10^{76}$
real *4 (MIL 1750A)
24 8
-2^0 2^{-1} 2^{-2} fraction 2^{-23} -2^7 2^6 exp 2^0
B_{31} B_8 B_0
$v = \text{fff} \times 2^e$
$\pm 1.5 \times 10^{-39}$ to $\pm 1.7 \times 10^{38}$

Figure 10.24. Number formats.

zeros, +0 and −0), infinity (e = all 1s, fff = 0; therefore both signs), and a curious class of reserved quantities known officially as NANs (NAN = "not a number")

The other important microcomputer floating-point format is DEC's, used in the MicroVAX and LSI-11 computers (and their ancestors, the VAX and PDP-11 minicomputers). It is very close to the IEEE standard, with the same number of exponent and mantissa bits (including the use of a hidden bit) used for single-precision numbers. In fact, the only differences are the exponent bias (128 instead of 127) and the fact that the mantissa has no leading bits, being instead of the form .1fff (with the "1" hidden). DEC defines only one zero (all bits zero), and does not permit non-normalized numbers or infinity; there are, however, analogs of the IEEE NANs. DEC also has a 64-bit double-precision format.

The last two formats in Figure 10.24 are used in large or special-purpose computers, but not in microcomputers. The "IBM" format has been used in mainframe IBM computers for some time and even in minicomputers like the Nova line from Data General. The 7-bit biased exponent tells the power of 16, rather than 2, giving greater exponent range. The mantissa therefore may have up to three leading zeros; i.e., a normalized fraction has a nonzero most significant hex digit.

EXERCISE 10.10

In order to understand the meaning of this last statement, write out the IBM representation of the number 1.0. Now write the next smaller number that can be represented in this format.

By its choice of exponent radix the IBM format sacrifices some precision for dynamic range. Furthermore, the precision varies somewhat from one number to another, owing to the variable number of leading binary zeros; this is known as "wobble." IBM format has no infinities or NANs, and only one zero (all bits zero); it does permit non-normalized numbers. IBM also has a 64-bit double-precision format.

The last format in the figure is MIL-STD-1750A, used in military systems. It is unusual in departing from the "sign/magnitude" convention of the previous formats, using instead a 2's complement mantissa with a 2's complement exponent. (Actually, the previous formats are more accurately described as sign/magnitude mantissa with offset-binary exponents.) It has no infinities, NANs, or non-normalized numbers; it, too, has a double-precision version.

Number storage in memory

Microprocessor designers like to express their individuality by storing numbers in memory in peculiar orders. The 8086/8 (therefore the IBM PC and compatibles) stores numbers beginning with the least significant byte in the lowest-numbered memory byte; the 68000 family does it the other way around. Lots of luck!

I/O data conversion

We detoured earlier to discuss number formats in the context of our hardware interface with its packed-BCD format. What is the best way to handle the kind of 8-digit data you would get from such an interface? Depending on the type of input data, the number of significant digits, its range of variation, etc., it may be best to convert the incoming data to floating-point (for greatest dynamic range) or to integers (for best resolution) or to do some other sort of numerical massaging (e.g., taking differences from the average value, or between successive data). This might be done in the particular device's software "driver," the section of program that handles the actual input of data. In this sense the software cannot be optimized without an understanding of the hardware and what its data means. Just another reason why it is important to know your way around the wonderful world of electronic hardware!