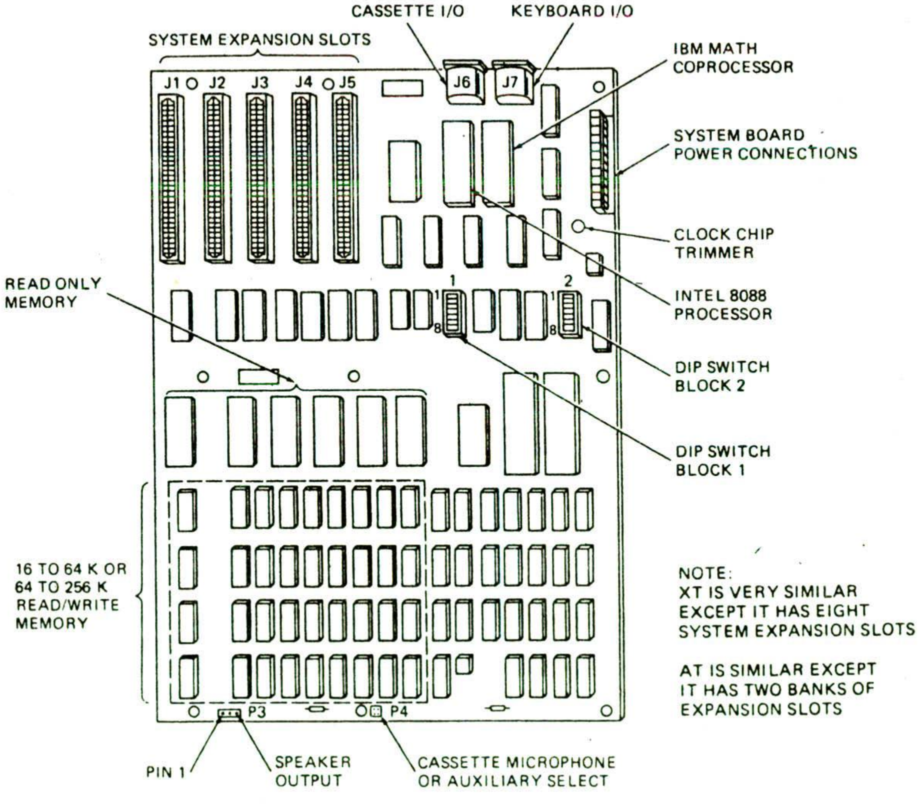
CHAPTER 11

DMA, DRAMs, Cache Memories, Coprocessors, and EDA Tools

The major objective of the first six chapters of this book was to introduce you to structured programming and to writing 8086 assembly language programs. Chapters 7 through 10 introduced you to the hardware of an 8086 minimum-mode system, showed you how to interface a microcomputer to a wide variety of input and output devices, and finally demonstrated how all these pieces are put together to build a simple microcomputer-based instrument or control system. The major goal of the remaining chapters in the book is to show you the hardware and software of larger microcomputer systems.

As an example of what we mean by a larger system, look at Figure 11-1, which shows the component side of the main microprocessor board or "motherboard" for an IBM PC. As you can see, the board contains an 8088 microprocessor, ROM, and a large block of dynamic RAM. The board also has a socket for a special 8087 math auxiliary processor. Finally, note the system expansion slots in the upper left corner of Figure 11-1. These slots allow you to plug in additional boards which give the system the specific interface functions you need. For example, you may want to add a disk-controller board,



NOTE:
XT IS VERY SIMILAR EXCEPT IT HAS EIGHT SYSTEM EXPANSION SLOTS

AT IS SIMILAR EXCEPT IT HAS TWO BANKS OF EXPANSION SLOTS

FIGURE 11-1 Component layout diagram for IBM PC motherboard. *(IBM Corporation)*

a serial-port board, a CRT controller board, a board with additional memory, an A/D-D/A board, or a board which allows your PC to function as a logic analyzer. This "open-system" approach lets you easily customize a system for your applications and financial state.

In later chapters we discuss the operation of peripheral boards such as CRT controller boards, disk-drive controller boards, and serial communication boards which plug into these expansion slots.

The first goal of this chapter is to show you how the circuitry on a microcomputer motherboard such as the one in Figure 11-1 works. A second goal of this chapter is to show you how computer-based tools are used to design, test, debug, and produce the hardware and software for a board such as this.

OBJECTIVES

At the end of this chapter you should be able to:

1. Show how an 8086 is connected with a controller device for operation in its maximum mode.
2. Show how a direct memory access (DMA) controller device can be connected in an 8086 system and describe how a DMA data transfer takes place.
3. Describe how large banks of dynamic RAM can be connected in a system.
4. Describe how a cache memory is used to reduce the number of wait states required in a system which has a large dynamic RAM main memory.
5. Describe how automatic error detecting-correcting circuitry works with memories.
6. Show how a coprocessor can be connected to an 8086 or 8088 operating in maximum mode.
7. Describe how an 8086 and an 8087 cooperate during the execution of a program which contains instructions for each.
8. Write a simple assembly language program for an 8087.
9. Describe how schematic capture programs, simulator programs, and other computer-based tools are now used to develop a microcomputer system.

INTRODUCTION

After much agonizing we finally decided to use the original IBM PC for some of the system examples in this chapter. Although the PC itself is somwhat outdated, it demonstrates well the concepts of DMA, DRAM interface, and coprocessors that we want to teach here. Almost all our discussion of PC operation is also valid for a PC/AT, and as you will see in later chapters, an understanding of the basic PC is a good starting point for understanding later generation systems.

To give you a more detailed idea of where we are going in this chapter and how it relates to what you have learned in previous chapters, let's take a look at Figure 11-2, which shows a block diagram of circuitry on an IBM PC motherboard. As you look at this diagram you should see many familiar parts and a few new ones. Start on the left side of the diagram and work your way across it from the 8088 CPU and the 8259A priority-interrupt controller. Under the 8088 main processor note the auxiliary processor socket which can be used for an 8087 math coprocessor.

The next vertical line of devices to the right in Figure 11-2 consists of the address bus buffers, the data bus buffers, and the 8288 bus controller chip. As we explain later, a bus controller chip is required to generate control bus signals when the 8088 is operated in its maximum mode. The buses from these devices go across the drawing and connect to the 62-pin peripheral board connectors so the 8088 can communicate with the boards in the peripheral expansion slots as well as with the ROM, RAM, and ports on board. Incidentally, the layout of the IBM PC/XT motherboard is very similar to this layout, but the XT has eight I/O slots instead of five.

Now find the ROM in the lower center, the keyboard logic, etc., in the middle right, and the dynamic RAM in the upper right. Finally, take a look at the column of devices which contains the 8237A-5 DMA controller. Starting at the bottom of this column you see an 8253-5 programmable timer which is nearly identical to the 8254 we described in Chapter 8. Just above this is the familiar 8255A-5 programmable port device. Now you are left to ponder just the three devices with DMA in their labels.

The major parts of this circuit that are new to you are the DMA section, the dynamic RAM section and its associated parity check/generator logic, and the auxiliary processor. In the following sections of the chapter we discuss each of these types of circuitry in detail. First, however, we will explain what we mean when we say that an 8086 or 8088 is operating in maximum mode because many of the circuits shown in this chapter and the following chapters use the devices in this mode.

THE 8086 MAXIMUM MODE

Figure 11-3*a*, p. 348, shows the pin diagram of the 8086 again. You may remember from our discussion in Chapter 7 that if pin 33, the MN/$\overline{\text{MX}}$ pin, is tied high, the 8086 operates in its minimum mode. In minimum mode the 8086 directly generates the control bus signals shown in parentheses next to pins 24 through 31 in Figure 11-3*a*.

If the MN/$\overline{\text{MX}}$ pin is tied low, the 8086 operates in its maximum mode and pins 24 through 31 generate the signals named next to the pins in Figure 11-3*a*. In maximum mode the control bus signals are sent out in coded form on the status lines, $\overline{\text{S0}}$, $\overline{\text{S1}}$, and $\overline{\text{S2}}$. As shown in Figure 11-3*b*, an external controller device such as the Intel 8288 is used to produce the required control bus signals from these lines. Figure 11-3*b* shows the expanded names for each of the control bus signals generated by the 8288. Note in Figure 11-3*b* that 8282 octal latches are used to demultiplex the address signals

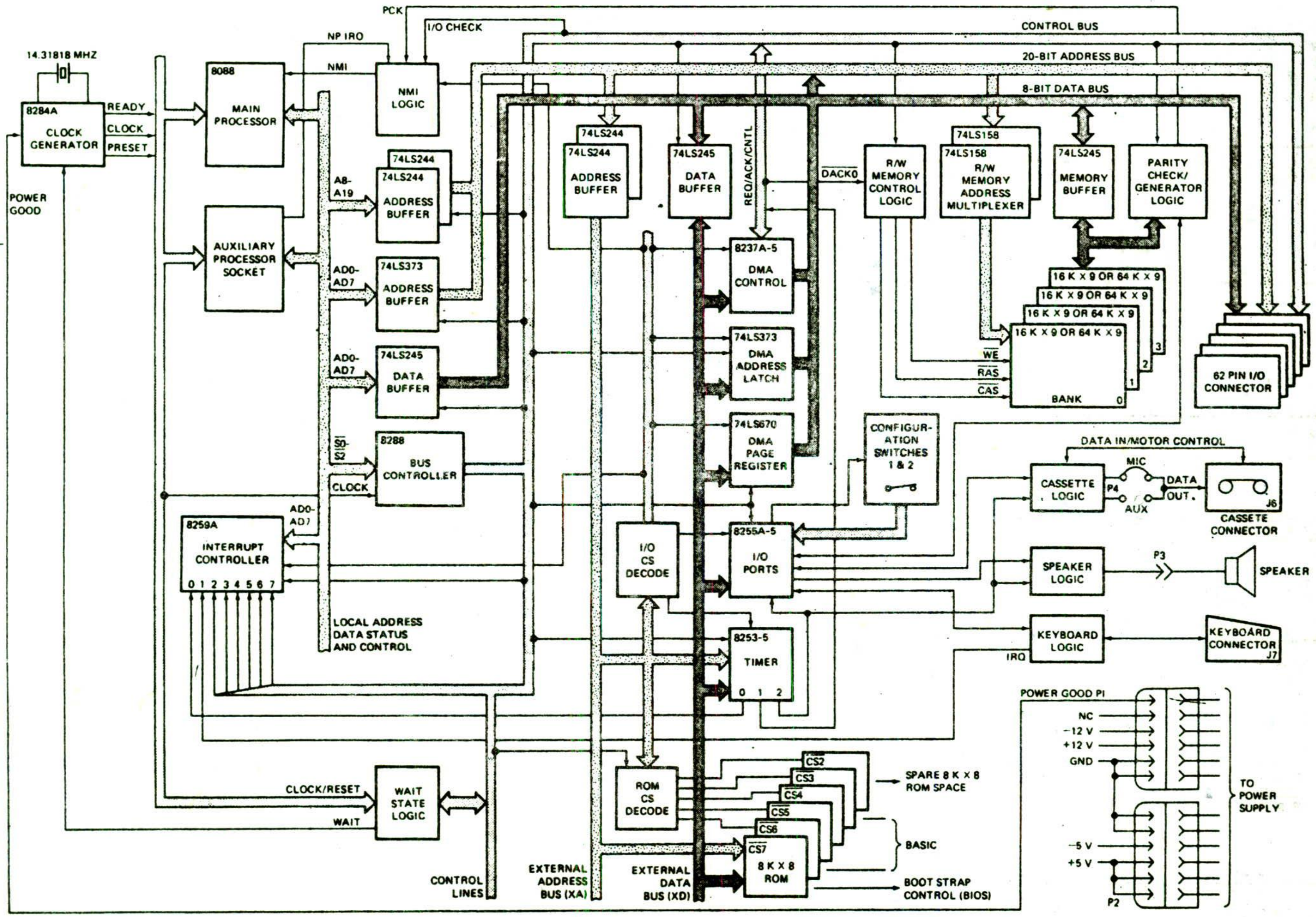


FIGURE 11-2 Block diagram of circuitry on IBM PC motherboard. (*IBM Corporation*)

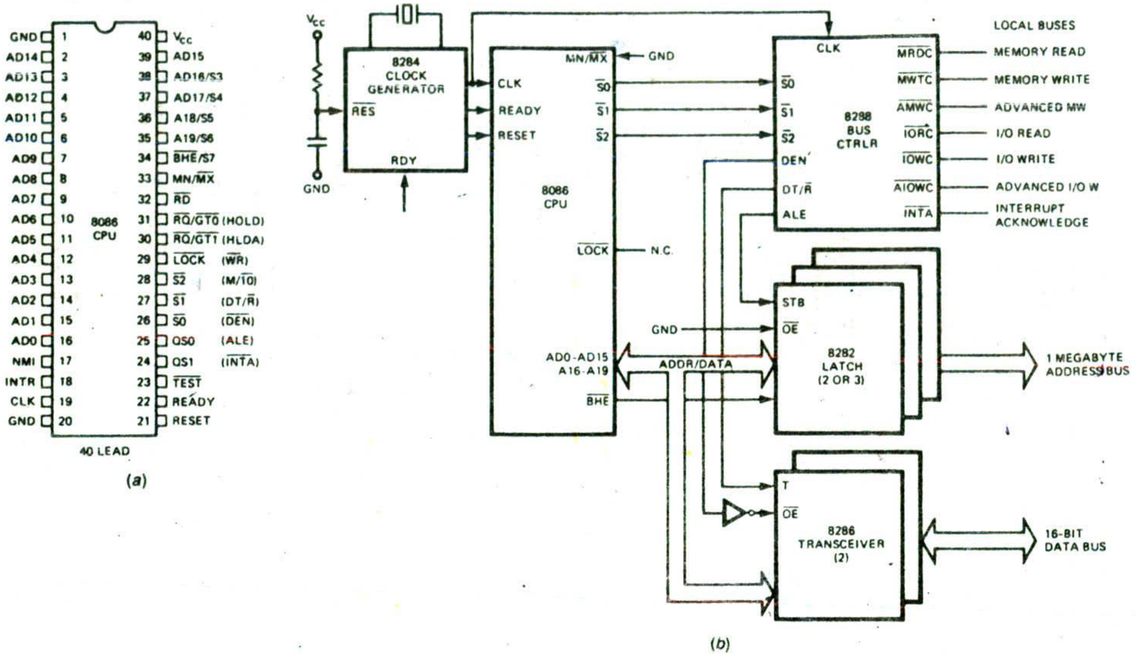


FIGURE 11-3 8086 revisited. (*a*) 8086 pin diagram. (*b*) Circuit showing 8086 connections for MAX mode operation. (*Intel Corporation*)

and 8286 bidirectional drivers are used to buffer the data bus so that it can drive a boardful of devices.

Now we will show you some of the ways that a microprocessor can timeshare its buses in minimum mode and maximum mode.

DIRECT MEMORY ACCESS (DMA) DATA TRANSFER

DMA Overview

Up to this point in the book we have used program instructions to transfer data from ports to memory or from memory to ports. For some applications, such as transferring data bytes to memory from a magnetic or optical disk, however, the data bytes are coming in from the disk faster than they can be read in with program instructions. In a case like this we use a dedicated hardware device called a *direct memory access* or *DMA controller* to manage the data transfer. The DMA controller temporarily borrows the address bus, data bus, and control bus from the microprocessor and transfers the data bytes directly from the disk controller to a series of memory locations. Because the data transfer is handled totally in hardware, it is much faster than it would be if done by program instructions. A DMA controller can also transfer data from memory to a port. Some DMA devices even can do memory-to-memory transfers to implement fast block transfers. Here's an example of how a common DMA controller is connected and used in an 8086 minimum-mode system.

Circuit Connections and Operation of the Intel 8237 DMA Controller

We chose the 8237 DMA controller as the example for this section because it is a commonly used device; also, it is one of the devices you will find if you start poking around inside an IBM PC/XT or PC/AT. Before we dig into the actual connections and operation of an 8237 circuit, however, let's take a look at the block diagram in Figure 11-4 to get an overview of how a DMA transfer takes place. The main point to keep in your mind here is that the microprocessor and the DMA controller timeshare the use of the address, data, and control buses. The three switches in the middle of the block diagram are an attempt to show how control of the buses is transferred.

When the system is first turned on, the switches are in the up position, so the buses are connected from the microprocessor to system memory and peripherals. We initialize all the programmable devices in the system and go on executing our program until we need, for example, to read a file off a disk. To read a disk file we send a series of commands to the smart disk controller device, telling it to find and read the desired block of data from the disk. When the disk controller has the first byte of data from the disk block ready, it sends a *DMA request*, DREQ, signal to the DMA controller. If that input (channel) of the DMA controller is unmasked, the DMA controller will send a *hold-request*, HRQ, signal to the microprocessor HOLD input. The microprocessor will respond to this input by floating its buses and sending out a *hold-acknowledge* signal, HLDA, to the

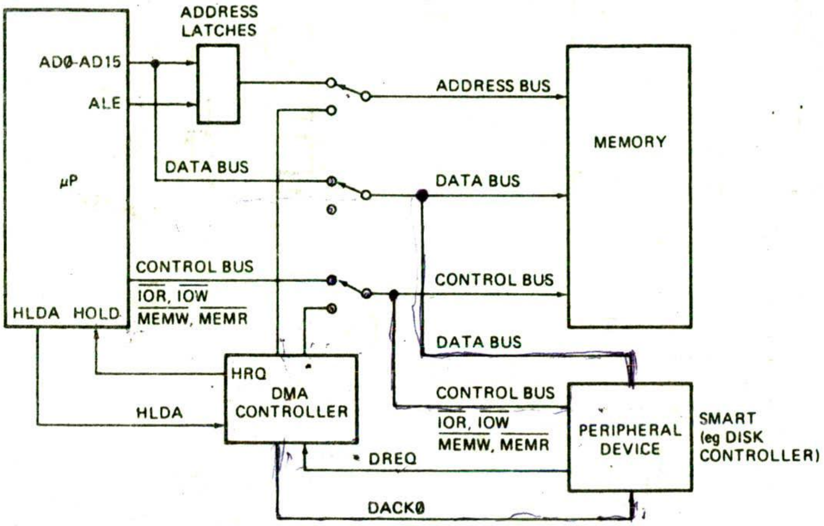


FIGURE 11-4 Block diagram showing how a DMA controller operates in a microcomputer system.

DMA controller. When the DMA controller receives the HLDA signal, it will send out a control signal which throws the three bus switches down to their DMA position. This disconnects the processor from the buses and connects the DMA controller to the buses.

When the DMA controller gets control of the buses, it sends out the memory address where the first byte of data from the disk controller is to be written. Next the DMA controller sends a *DMA-acknowledge*, DACKO, signal to the disk controller device to tell it to get ready to output the byte. Finally, the DMA controller asserts both the $\overline{\text{MEMW}}$ and the $\overline{\text{IOR}}$ lines on the control bus. Asserting the $\overline{\text{MEMW}}$ signal enables the addressed memory to accept data written to it. Asserting the $\overline{\text{IOR}}$ signal enables the disk controller to output the byte of data from the disk on the data bus. The byte of data then is transferred directly from the disk controller to the memory location without passing through the CPU or the DMA controller.

NOTE: For this type of transfer the disk controller chip select input does not have to be enabled by the port address decoding circuitry as it does for normal reading from and writing to registers in the device. In fact, the normal port-decoding circuitry is disabled during DMA operations to prevent the combination of $\overline{\text{IOR}}$ and the output memory address from turning on unwanted ports.

When the data transfer is complete, the DMA controller unasserts its hold-request signal to the processor and releases the buses. The switches in Figure 11-4 are effectively thrown back up to the CPU position. This lets the processor take over the buses again until another DMA transfer is needed. The processor continues executing from where it left off in the program.

A DMA transfer from memory to the disk controller proceeds in a similar manner except that the DMA controller asserts the memory-read control signal, $\overline{\text{MEMR}}$, and the output-write control signal, $\overline{\text{IOW}}$. DMA transfers may be done a byte at a time or in blocks.

Now, to give you more practice working your way through actual microprocessor circuits, let's look at Figure 11-5, p. 359, to see some of the circuitry we might add to an 8086 system so that we can do DMA transfers to and from a disk controller. This circuitry is simply a more detailed version of the block diagram in Figure 11-4.

The first thing to do in analyzing this schematic is to identify the major devices and relate their function, where possible, to the block diagram. The 8086 and 8284 should be old friends from your exploration of the SDK-86. The 8237 is, of course, the DMA controller, and the 8272 is the floppy-disk controller. We discuss the operation of a disk controller more in Chapter 13, but for now all you need to know about it is the overview of how it interacts with the 8237, as we described earlier. The 8282s in this circuit are octal latches with three-state outputs. They are used here to latch addresses output from either the 8086 or from the DMA controller. These devices are controlled by ALE from the 8086 and by AEN and ADSTB from the DMA controller.

When the power is first turned on, the *address-enable* signal, AEN, from the DMA controller is low. Devices U1, U2, and U4 are then enabled, and the ALE signal from the 8086 goes to the strobe inputs of all three devices. When the 8086 sends out an address and an ALE signal, these three devices will grab the address and send it out on the address bus lines, A19–A0. This is just as would be done in a simpler 8086 system. Now, when the DMA controller wants to take over the bus, it asserts its AEN output high. This does several things. First, it disables

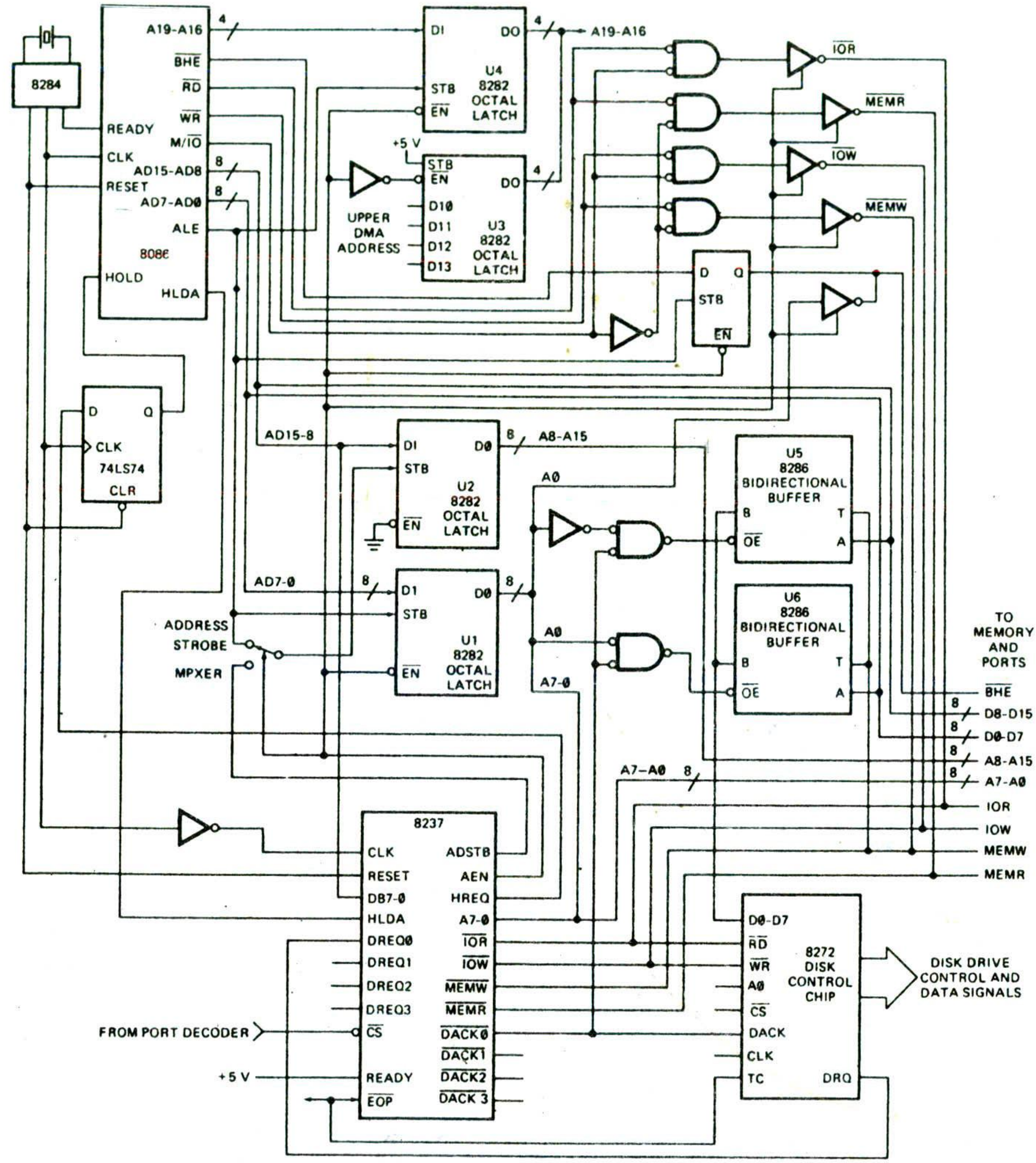


FIGURE 11-5 Schematic for 8086 system with 8237 DMA controller and 8272 floppy-disk controller.

device U1 so that address lines A7–A0 no longer come from the 8086 bus. The 8237 directly outputs the lower 8 bits of the memory address for the DMA transfer.

Second, AEN, going high, switches the strobe multiplexer so that the strobe for device U2 comes from the address strobe output of the 8237. To save pins, the 8237 outputs the upper 8 bits of the memory address for the DMA transfer on its data bus pins and asserts its ADSTB output high to let you know that this address is present there. At the start of a DMA transfer, then, memory address bits A15–A8 will be sent out by the 8237 and latched on the outputs of U2.

Still another effect of AEN going high is to switch the source of address bits A19–A16 from device U4 to device U3. The DMA controller does not send out these address bits during a DMA transfer, so you have to produce them in some other way. You can either hard-wire the inputs of U3 to ground or +5 V to produce a fixed value for these bits, or you can connect these inputs to an output port so you can specify these address bits under program control.

Finally, AEN going high switches the source of the control bus signals from the outputs of the control bus decoder circuitry to the control bus signal outputs of the DMA controller. This is necessary because, during a DMA transfer, the 8237 generates the required control bus signals such as $\overline{\text{MEMW}}$ and $\overline{\text{IOR}}$. Incidentally, the NOR gate decoder circuitry in the upper right corner of the schematic is necessary to produce processor control bus signals compatible with those from the 8237.

The final part of the circuit in Figure 11-5 to analyze is the two 8286 octal bus transceivers. The disk controller has bnly an 8-bit data bus output. If we had connected these eight lines on the lower eight data bus lines of the 8086 system, the DMA controller would be able to transfer bytes only to even addresses. Likewise, if we had connected the disk controller data outputs on the upper eight data lines of the 8086 system, the DMA controller would be able to transfer bytes only to odd addresses in memory. To solve this problem, we connect the two 8286s as a switch which can route data to/from the disk controller from/to either odd or even addresses in memory. If you work through the glue logic, you should see that A0 determines which half of the data bus is connected to the eight data pins of the disk controller. $\overline{\text{MEMW}}$ determines whether the buffers are set to transfer data to or from the disk controller. Now let's look more closely at the signal flow and timing for this circuit.

A DMA Transfer Timing Diagram

Figure 11-6 shows the sequence of signals that will take place for a DMA transfer in a system such as that in Figure 11-5. Keep a copy of Figure 11-5 handy as you work your way down through these waveforms. The labels we have added to each signal should help you. We will pick up where the 8237 asserts AEN high and gains control of the buses. After the 8237 gains control of the bus, it sends out the lower 8 bits of the memory address on its A7–A0 pins and the upper 8 bits of the memory address on its DB0–DB7 pins. The 8237 pulses ADSTB high to latch these address bits in the 8282 and then removes these address bits from the data bus. At about the same time thc 8237 sends a DACK signal to the disk controller to tell it to get ready for a data transfer.

Now that everything is ready, the 8237 asserts two control bus signals to enable the actual transfer. For a transfer from memory to the disk controller, it will assert $\overline{\text{MEMR}}$ and $\overline{\text{IOW}}$. For a transfer from the disk controller to memory, it will assert $\overline{\text{MEMW}}$ and $\overline{\text{IOR}}$. Note that the 8237 does not have to put out an I/O address to enable the disk controller for this transfer. When programmed in DMA mode, the disk controller needs only $\overline{\text{IOR}}$ or $\overline{\text{IOW}}$ to be asserted to enable it for the transfer. Also note that the 8237 will not output a new address on A8 through A15 when a second transfer is done, unless those bits have to be changed. This saves time during multiple-byte transfers.

When the programmed number of bytes have been transferred, the DMA controller pulses its end-of-proc-

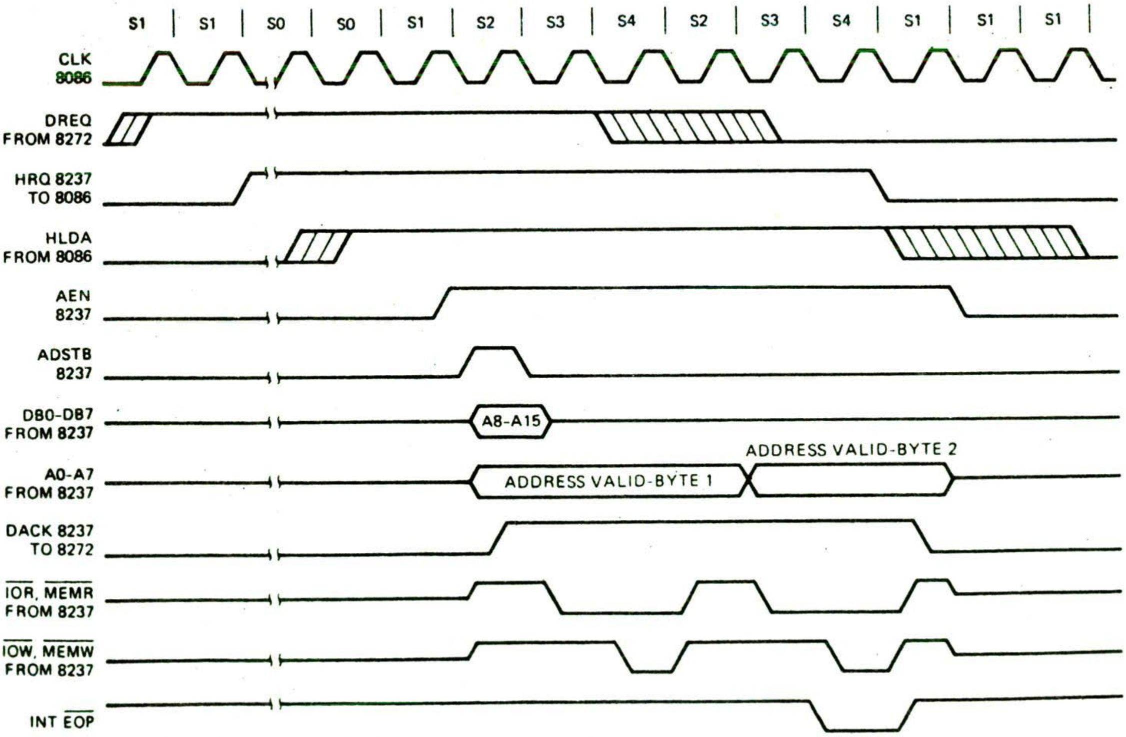


FIGURE 11-6 Timing diagram for 8237 DMA transfer. *(Intel Corporation)*

ess, $\overline{EOP}$, pin low, unasserts its hold request to the 8086, and drops its AEN signal low. This releases the buses back to the 8086. Now that you have an idea how an 8237 is connected and operates in a system, we will give you an overview of what is involved in initializing it.

8237 Initialization Overview

Initializing an 8237 is not difficult, but it does require a fairly large number of bytes. We do not have space to show you a complete initialization, but here is an overview.

The 8237 is connected in a system as a port device, so you write initialization words to it just as you would to any other port device. Incidentally, several 8237s can be cascaded in a master-slave arrangement to give more input channels and each device must be initialized.

As shown by the pin labels on the 8237 in Figure 11-5, the 8237 has four DMA request inputs or *channels*, as they are commonly called. For each channel you need to send a command word which specifies the general operation, mode words, the starting memory address, and the number of bytes to be transferred. Each channel of the 8237 can be programmed to transfer a single byte for each request, a block of bytes for each request, or to keep transferring bytes until it receives a wait signal on the $\overline{EOP}$ input/output. Consult the data sheet in an Intel data book to get the details of each command word.

DMA and the IBM PC

Now that you know how DMA operates, let's take a look back at the DMA section in the block diagram of the IBM PC motherboard circuitry in Figure 11-2. The 8237A-5 is, of course, the DMA controller. The 74LS373 just under it is used to grab the upper 8 bits of the DMA address sent out on the data bus by the 8237A-5 during a transfer. This device has the same function as device U2 in Figure 11-5. The 74LS670 just below this is used to output bits A16–A19 of the DMA transfer address, the same function performed by U3 in the circuit in Figure 11-5.

In order that peripheral boards can interface with the motherboard circuitry on a DMA basis, the DMA signal lines are connected to the peripheral connectors shown in the upper right corner of Figure 11-2. To see how DMA and other signals go to the peripheral boards, take a look at the pin descriptions in Figure 11-7.

The signals shown in Figure 11-7*a* are bused to all five peripheral connectors in parallel so that any board can access them. Most of the signals on these connectors

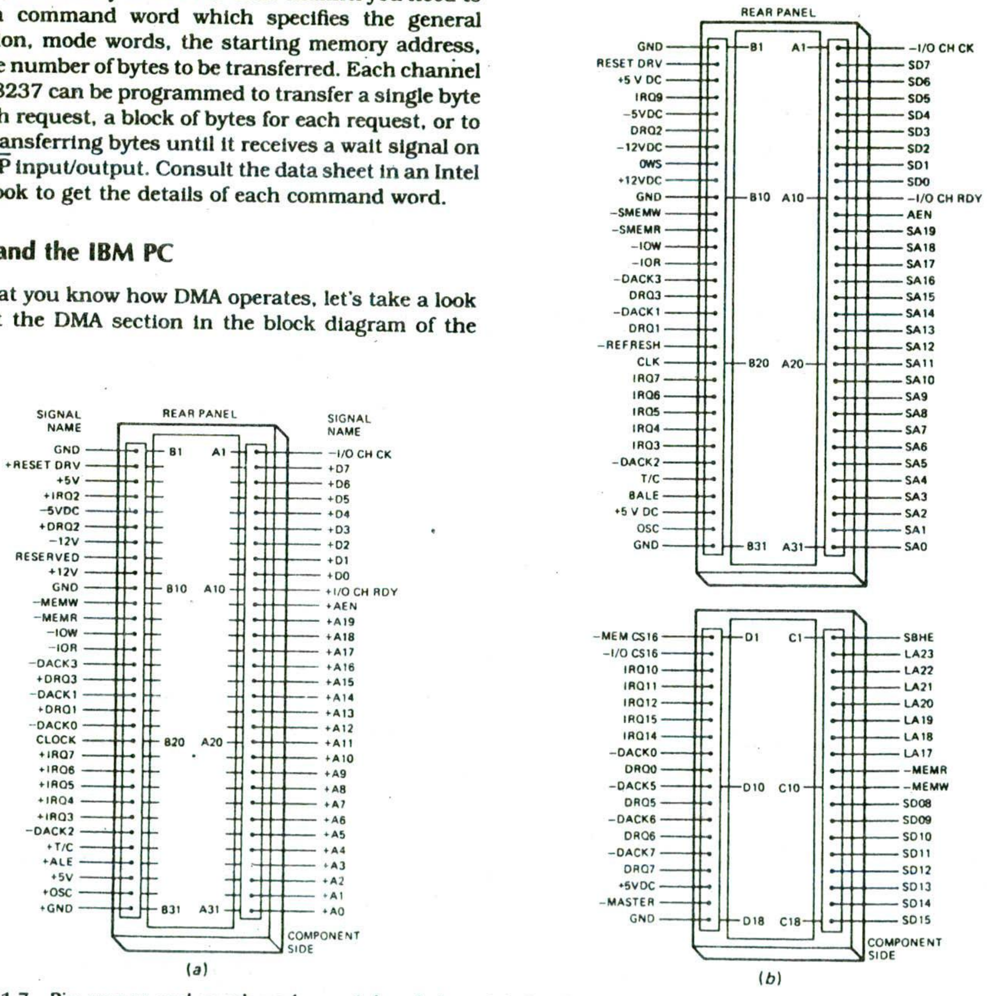


FIGURE 11-7 Pin names and numbers for peripheral slots. (a) On IBM PC motherboard. (b) On IBM PC/AT motherboard. *(IBM Corporation)*

should be easily recognizable to you. A + in front of a signal indicates that the signal is active high, and a − indicates that the signal is active low. A0 through A19 on the connectors are the 20 demultiplexed address lines, and D0 through D7 are the eight data lines. IRQ2 through IRQ7 are interrupt request lines which go to the 8259A priority-interrupt controller so that peripheral boards can interrupt the 8086 if necessary. Some other simple signals on the connectors are the power supply voltages; the standard ALE, −MEMW, −MEMR, −IOW, and −IOR control bus signals; and some clock signals. The I/O CH RDY pin on the connector can be asserted by a peripheral board to cause the 8086 to insert WAIT states until the peripheral board is ready.

Finally, we are down to the DMA signals on the expansion connectors. The DMA request pins DRQ1–DRQ3 allow peripheral boards to request use of the buses. A disk controller board, for example, might request a DMA transfer of a block of data from system memory. When the DMA controller gains control of the system buses, it lets the peripheral device or board know by asserting the appropriate −DACK0 through −DACK3 signal. The AEN signal on the connectors is used to gate the DMA address on the bus, as we described earlier. When the programmed number of bytes has been transferred, the T/C pin on the connector goes high to let the peripheral know that the transfer is complete.

To show you that it is a small step to understand another bus, Figure 11-7*b* shows the pin names for the I/O bus connectors on IBM PC/AT. The 80286 microprocessor used in the AT has 24 address lines which are sent out on the bus as SA0 through SA19 and LA20 through LA23. SD0 through SD15 are the 16 data lines for the bus. The AT motherboard uses two 8259A priority interrupt controllers to produce 15 interrupt inputs. The 11 interrupt inputs not used in the motherboard are connected to IRQ pins on the bus. The AT uses two 8237A DMA controllers to produce 7 DMA channels. The DREQ input signal for each channel and the DACK signal for each channel are present on the bus. Other DMA signals present are AEN and T/C. The next signals to look for in Figure 11-7*b* are the control bus signals, −SMEMW, −SMEMR, −IOW, −IOR, −MEMW, −MEMR, BALE, and SBHE, which should be fairly familiar to you from our previous discussions of the 8086. Now, all you have left are a few miscellaneous signals such as REFRESH, which is used to indicate a DRAM refresh operation is in process; −MEM CS16, which lets the motherboard know that the present data transfer is a 1 wait-state transfer; and −I/O CS16, which is used to let the motherboard know that the present data transfer is a 1 wait-state transfer. The 0WS line on the bus is used to tell the motherboard that no wait states are required to complete the current read or write cycle. RESET DRV is the system reset line, CLK is the 6.0-MHz system clock, and OSC is a high frequency clock signal which can be divided down and used on I/O boards. Finally, the −MASTER signal is used by another processor board to gain control of the bus.

Now that you know how DMA works in a microcomputer, the next block of circuitry to talk about is the RAM section.

INTERFACING AND REFRESHING DYNAMIC RAMs

Review of Dynamic RAM Characteristics

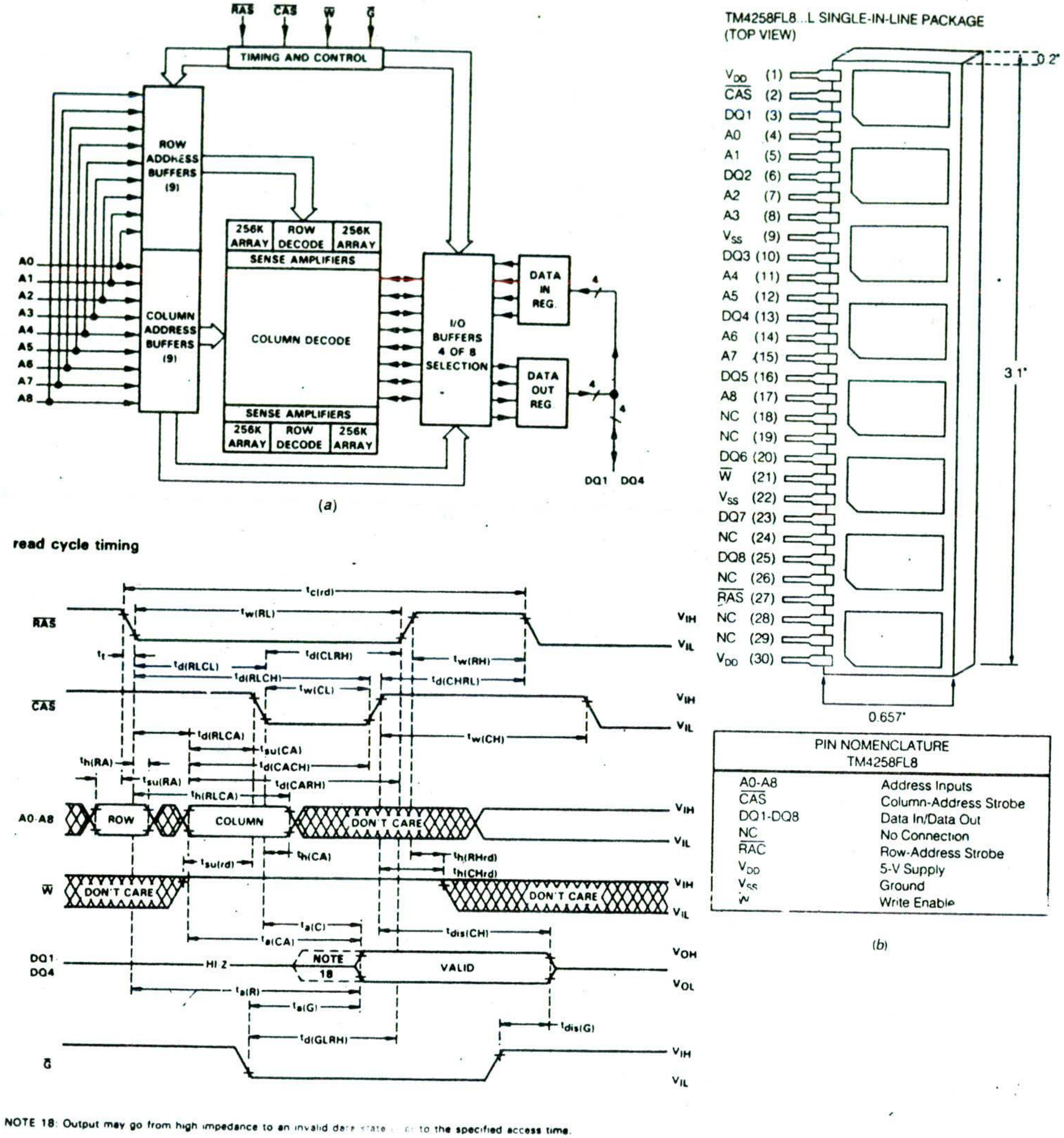
For small systems such as the SDK-86, where we only need a few kilobytes of RAM, we usually use static RAM devices because they are very easy to interface to. For larger systems, where we want several hundred kilobytes or megabytes of memory, we use dynamic RAMs, often called DRAMs. Here's why.

Static RAMs store each bit in an internal flip-flop which requires four to six transistors. In DRAMs a data bit is stored as a charge or no charge on a tiny capacitor. All that is needed in addition to the capacitor is a single transistor switch to access the capacitor when a bit is written to it or read from it. The result of this is that DRAMs require much less power per bit, and many more bits can be stored in a given size chip. This makes the cost per bit of storage much less. The disadvantage of DRAMs is that each stored data bit must be refreshed every 2 to 8 ms because the charge stored on the tiny capacitors tends to change due to leakage. When activated by an external signal, the refresh circuitry in the device checks the voltage level stored on each capacitor. If the voltage is greater than $V_{CC}/2$, then that location is charged to V_{CC}. If the voltage is less than $V_{CC}/2$, then that location is discharged to 0 V. Let's take a look at a typical DRAM to see how we read, write, and refresh it.

Figure 11-8*a*, p. 354, shows an internal block diagram for a Texas Instruments TMS44C256 CMOS DRAM. This device is a 256K × 4 device, so it stores 262,144 words of 4 bits each in its 20-pin package. You can connect two of these in parallel to store bytes or 4 in parallel to store 16-bit words. Since DRAMs are almost always connected in parallel, several companies now produce DRAM modules such as the TI TM4256FL8 256K × 8 device shown in Figure 11-8*b*. The 30-pin single in-line package (SIP) takes much less PC board space than the equivalent DIPs.

Now, according to the basic rules of address decoding, 18 address lines should be required to address one of the 256K or 2^{18} words stored in the MT44C256 DRAM. The diagram in Figure 11-8*a*, however, shows only nine address inputs, A0–A8. The trick here is that to save pins, DRAMs usually multiplex in the address one-half at a time. A look at the timing diagram for a read operation in Figure 11-8*c* should help you to see how this works.

To read a word from a bank of dynamic RAMs, a DRAM controller device or other circuitry asserts the write-enable, $\overline{W}$, pin of the DRAMs high to enable them for a read operation. It then sends the upper half of the address, called the row address or page address, to the nine address inputs of the DRAMs. The controller then asserts the *row-address-strobe*, $\overline{RAS}$, input of the DRAM low to latch the row address in the DRAM. After the proper timing interval, the controller removes the row address and outputs the lower half of the address, called the column address, to the nine address inputs of the DRAMs. The controller then asserts the *column-*



PIN NOMENCLATURE TM4258FL8	
A0-A8	Address Inputs
$\overline{CAS}$	Column-Address Strobe
DQ1-DQ8	Data In/Data Out
NC	No Connection
$\overline{RAC}$	Row-Address Strobe
V_{DD}	5-V Supply
V_{SS}	Ground
$\overline{W}$	Write Enable

(b)

NOTE 18: Output may go from high impedance to an invalid data state ... to the specified access time.

FIGURE 11-8 TMS44C256 D k diagram. (b) 30-pin SIP diagram. (c) Read-cycle ti

address-strobe, $\overline{CAS}$ the column address s. After a delay, the data wor the addressed memory will appear on the da utputs of the DRAMs.

The timing diagra r a write cycle except that after it sends out the column address and $\overline{CAS}$ the controller asserts the write-enable, $\overline{W}$, input able the DRAMs for writing, and asserts a signal sed to gate the data to be written onto the f the DRAMs.

a row in a DRAM, the row address is applied ess inputs and the $\overline{RAS}$ input is pulsed low. For this particular device each row must be refreshed at least once every 8 ms. The refresh can be done in either a *burst* mode or in a *distributed* mode. In the burst mode all 512 rows are addressed and pulsed with a $\overline{RAS}$ strobe one right after the other every 8 ms. In the distributed mode a row is addressed and pulsed after every 8/512 ms or 15.6 µs. In a particular system you use the mode which least interferes with the operation of the system. Now that the operation of dynamic RAMs is fresh in your mind, we will show you how you interface banks of DRAMs to an 8086.

Overview of Interfacing DRAMs to a Microprocessor

As perhaps you can see from the preceding discussion, the following are the main tasks you have to do to interface a bank of DRAMs to a microprocessor:

1. Multiplex the two halves of the address into each device with the appropriate $\overline{RAS}$ and $\overline{CAS}$ strobes.
2. Provide a read/write control signal to enable data into or out of the devices.
3. Refresh each row at the proper interval.
4. Ensure that a read or write operation and a refresh operation do not take place at the same time.

There are many ways to do these tasks. For a start let's look at how it is done in an IBM PC or PC/XT type microcomputer.

DRAM Interfacing and Refreshing in the IBM PC

As you can see in Figure 11-2, the IBM PC has four banks of 64K × 1 DRAMs. Two 74LS158 multiplexer devices are used to separate the two halves of an address as needed by the DRAMs. Some simple control logic generates the $\overline{RAS}$, $\overline{CAS}$, and RD/$\overline{W}$ signals. To refresh the DRAMs on the PC and PC/XT, we use a dummy DMA read approach. Here's how it works.

An 8253 timer is programmed to produce a pulse every 15 µs. This pulse is connected into one of the DMA request inputs (DREQ0) of an 8237 DMA controller, which has been programmed to read from memory and write to a nonexistent port. When the 8237 DMA controller receives this pulse, it sends a hold request to the 8088 microprocessor. After the 8088 responds with an HLDA signal, the 8237 takes over the buses, sends out a memory address, sends out a memory-read signal, and sends out a DMA acknowledge (DACK0) signal. The lower 8 bits of the memory address it sends out go to the address inputs of all of the DRAMs. The DACK0 signal from the DMA controller generates a signal which pulses the $\overline{RAS}$ lines of all of the DRAM banks low at this time. After each DMA operation the current address register in the DMA controller will be automatically incremented or decremented, depending on how the device was programmed. In either case, the next DMA operation will refresh the next row in the DRAMs. If the 8237 is programmed for transfer of 64 Kbytes, start at address 0, increment count after DMA, and autoinitialize; the sequence of addresses sent out will refresh all 256 rows in the DRAMs over and over. One row in each of the banks then is refreshed every 15 µs. With the 4.77-MHz clock used in the basic IBM PC, a refresh DMA cycle takes about 820 ns every 15 µs or about 5 percent of the processor's time.

The DRAM-refresh method used in IBM PC/AT-type microcomputers is not based on DMA, but it does put the microcomputer out of action for about 5 percent of the time. In a system where we don't want to sacrifice 5 percent of the processor's time for simply refreshing DRAMs, we use a dedicated controller device to do the refresh, etc. Here's an example of this type of device.

Using an 82C08 DRAM Controller IC with an 8086

In high-performance systems where we want DRAM refreshing to take up a minimum amount of the processor's time, we usually use a dedicated device which handles all of the refreshing chores without tying up the microprocessor or its buses as the DMA approach does. An example of this type of device is the Intel 82C08. Figure 11-9, p. 356, shows, in block diagram form, how an 82C08 can be connected with an 8086 in maximum mode to refresh and control 512 Kbytes of dynamic RAM. The 82C08 takes care of all of the addressing and refresh tasks we described before.

The memories here are the 256K × 4 devices shown in Figure 11-8*a*. As usual for an 8086 system, the memory is set up as 2-byte-wide banks. In this system each bank has two DRAM devices, so each bank has 256 Kbytes.

One important point to observe here is that the status signals, S0–S3, from the 8086 are connected directly to the control inputs of the 82C08. The 82C08 decodes these status signals to produce the read and write signals needed for the DRAMs. This advanced decoding means that, except when a refresh cycle is in progress, the 8086 will be able to read a byte or word from the DRAMs without WAIT states.

If you look closely at the 82C08 in Figure 11-9, you should find the port enable input, $\overline{PE}$. This input is asserted low to request access to the DRAM. If the 82C08 is not involved in a refresh operation when $\overline{PE}$ is asserted low, the 82C08 will multiplex the address from the address bus into the DRAMs with the appropriate $\overline{RAS}$ and $\overline{CAS}$ strobes. The 82C08 will also send out an $\overline{AACK}$ signal which clocks the 74LS74 flip-flops to transfer the A0 and $\overline{BHE}$ signals to the two memory banks. For a read operation the addressed byte or word will then be output on the data bus to the 8086. For a write operation the byte or word on the data bus will be written to the addressed locations in the DRAMs.

The output of an address decoder is connected to the $\overline{PE}$ input to assert it for the desired range of addresses. Because the DRAM banks in the circuit in Figure 11-9 are so large, the address decoding is very simple. Each bank in the circuit contains 256 Kbytes. Since 256K = 2^{18}, 18 address lines are required to address one of the bytes in a bank. In most systems we connect system address lines A1 through A18 to the 82C08 address inputs and the 82C08 multiplexes these signals

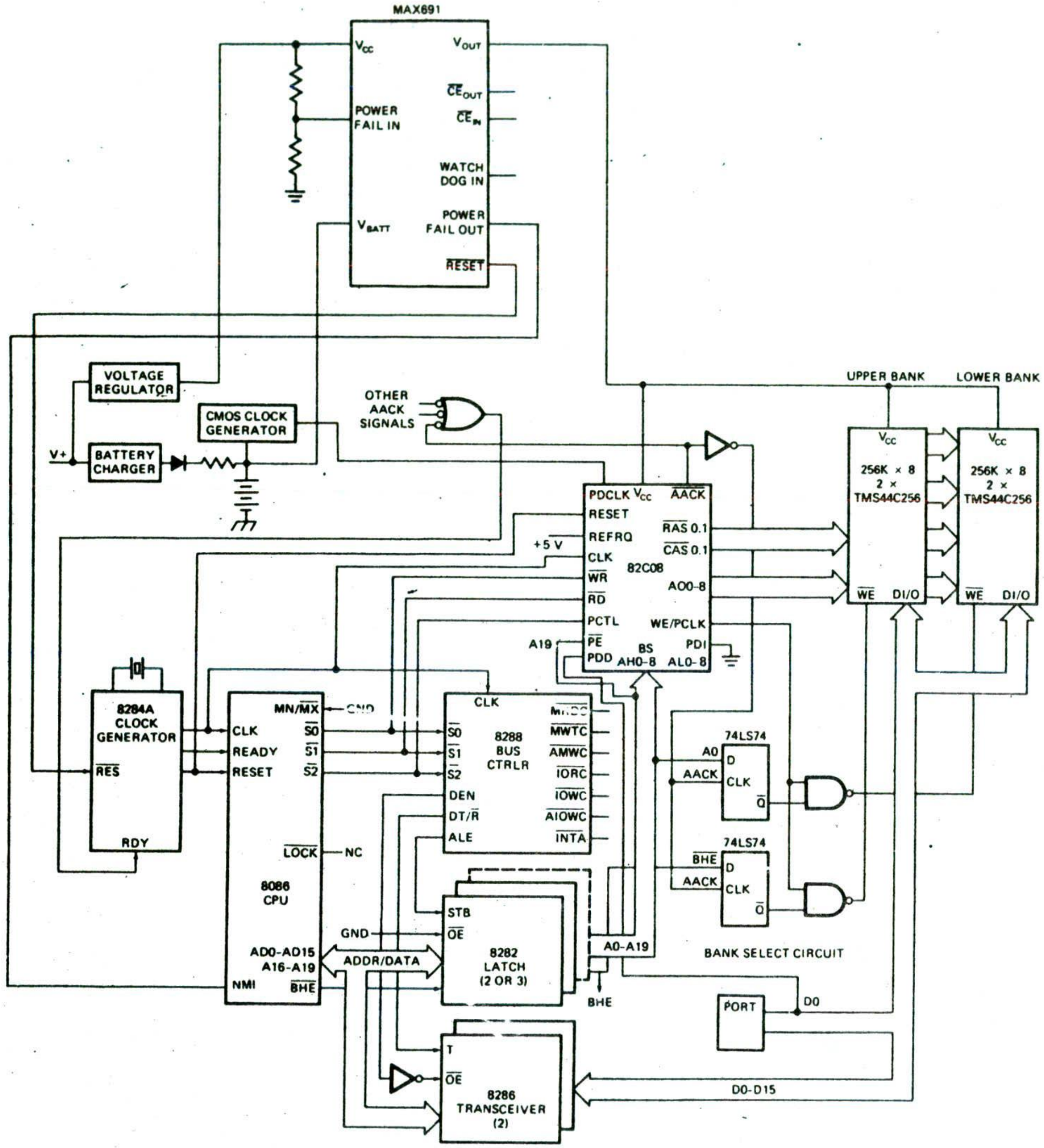


FIGURE 11-9 The 8086 microcomputer system using 82C08 DRAM controller.

into the DRAMs, nine at a time. Address line A0 is used along with the BHE signal to select the desired bank(s). This leaves only the A19 system address line unaccounted for. If we connect the A19 address line directly to the PE input of the 82C08, then PE will be asserted whenever the 8086 outputs a memory address with A19 low. In other words, the PE input will be asserted when the 8086 outputs any address between 00000H and 7FFFFH.

NOTE: The status signals from the 8086 are decoded in the 82C08, so it knows whether an address is intended for memory or an I/O port.

The address decoder here is simply a piece of wire or circuit trace which connects A19 to the PE input. This connection puts the RAM in the lower half of the 8086 address range, which is appropriate, because for an

8086 we want ROMs containing the startup program to be at the top of the address range.

The next point to consider in the system in Figure 11-9 is how the controller arbitrates the dispute that occurs if the CPU tries to read from or write to memory while the controller is doing a refresh cycle. If the 82C08 in Figure 11-9 happens to be in the middle of a refresh cycle when the 8086 tries to read a DRAM location, the 82C08 will hold its $\overline{\text{AACK}}$ high until it is finished with the refresh cycle. With the connections shown in Figure 11-9, this will cause the 8086 to insert one or more WAIT states while the 82C08 finishes its refresh cycle. In this system then, the occasional access conflict is arbitrated by the DRAM controller. Inserting a wait state now and then slows the 8086 down less than the DMA approach used in the IBM PC/XT-type computers.

Another interesting feature of the system in Figure 11-9 is the battery-backup circuitry. In Chapter 8 we discussed the use of an 8086 NMI interrupt procedure to save program data in the case of a power failure. In the few milliseconds between the time the ac power goes off and the time the dc power drops below operating levels, an interrupt procedure copies program data to a block of CMOS static RAM which has a battery-backup power supply. When the system is repowered, the saved data is copied back into the main RAM, and processing takes up where it left off. In larger systems there may not be time enough to copy all of the important data to another RAM, so we simply use a battery backup for the entire RAM array, as shown in Figure 11-9.

In this circuit we used CMOS DRAMs, because when these devices are not being accessed for reading, writing, or refreshing, they take only microwatts of power. During battery backup of the DRAMs they must still be refreshed, so the 82C08 DRAM controller is also connected to the battery power.

When the power supply voltage drops below a specified level, the PFO pin on the MAXIM 691 supervisor device sends a signal to the NMI input of the 8086. The NMI interrupt procedure saves parameters so the program can restart correctly when power returns and then sends a signal to the POWER DOWN DETECT (PDD) input of the 82C08. In response to this signal the 82C08 switches from the high-frequency system clock to a lower-frequency clock signal from the CMOS crystal oscillator. Reducing the clock frequency decreases the amount of current required by the DRAMs and by the 82C08 to perform refresh operations. Also, by using the CMOS oscillator, the high-current 8284 system clock generator does not need to be kept running.

When the power returns, the MAX691 generates a power-on-reset signal, $\overline{\text{RESET}}$, with the correct timing for the 8086. If a low is output to the PDD input of the 82C08 as part of the startup sequence, the 82C08 will automatically switch to using the system clock and operate normally for read, write, and refresh operations.

For the backup battery we use a nickel-cadmium or some other type which can stand the continuous recharging and supply the needed current. The diodes in the circuit prevent the power supply output and the battery from fighting with each other.

In applications where the entire system must be kept running during an ac power outage, we use a *noninterruptible power supply* or *NPS*. These power supplies contain large batteries, charging circuitry, and circuitry needed to convert the battery voltage to the voltages needed by the microcomputer.

Dynamic RAM Timing in Microcomputer Systems

In Chapter 7 we showed you how to determine if a memory device such as a ROM or RAM is fast enough to operate in an 8086 system with a given clock frequency. To make these calculations for a ROM or SRAM, you use its access times. For DRAMs, however, the limiting time is the read-cycle time, t_{RD}. Here's why.

If you take a close look at the read-cycle timing diagram for the TMS44C256 in Figure 11-8c, you should see that valid data will be present on the output a time $t_{a(R)}$ after $\overline{\text{RAS}}$ goes low. For the fastest current version of the device, this time is about 100 ns. Before another row in the device can be accessed, however, the $\overline{\text{RAS}}$ input has to be made high and held high for a time labeled $t_{w(RH)}$. This time of about 80 ns is required to *precharge* the DRAM so that it is ready to accept the next row address. (Reading data from a storage location in a row discharges that location somewhat and the internal circuitry in the DRAM "precharges" the location again before it allows access to another row.)

The precharge time effectively adds to the access time, so the time before a data bit from another row can be available on the output is considerably longer than the access time. The total time from the start of one read cycle to the start of the next is identified in Figure 11-8c as $t_{c(rd)}$. For the fastest version of the TMS44C256 the access time is only 100 ns, but the $t_{c(rd)}$ is 190 ns. For applications where the data words are rapidly being read from random rows, it is this $t_{c(rd)}$ that limits the rate at which words from random rows can be read. Let's see how this time fits in a microprocessor read cycle.

As shown in Figure 7-19, an 8086 requires four clock cycles for each memory access. If the 8086 is operated with a 10-MHz clock (100 ns per clock), a memory access cycle will take 400 ns. This means that if you are willing to pay the price, you can get DRAMs which will operate without wait states in a microcomputer using a 10-MHz 8086. However, as we discuss in Chapter 15, later-generation processors such as the 80386 require only two clock cycles for a memory access, and they are typically operated with a clock signal of 25 MHz or more. These factors drastically decrease the time available for memory access. If currently available DRAMs are used as the main memory in a microcomputer which has a clock frequency greater than about 15 MHz, one or more wait states must usually be inserted in every DRAM read or write cycle. However, the low cost per bit of DRAMs makes them attractive enough that several methods have been developed so they can be used without having to insert wait states in every memory access cycle. While the characteristics of DRAMs are fresh in your mind, we will introduce you to some of these techniques.

Page Mode and Static Column Mode DRAM Systems

Two of the most commonly used techniques to reduce the number of wait states needed with DRAMs are the *page mode* method and the *static column* method. Here's how they work.

Remember from our discussion of DRAMs in a previous section that a precharge time is required each time a new row (page) is accessed in a DRAM. This precharge time is the reason that the typical read and write cycle times are so much longer than the access times for DRAMs. If successive data words are read from or written to locations in the same page (row), however, no precharge time is required. Also, if successive data words are read from the same page, the row address is the same, so a new row address does not have to be sent out and strobed in with an $\overline{\text{RAS}}$ signal. With the proper DRAM controller these two factors make it possible to read data from a page or write data to a page without wait states. Some timing diagrams should help you see this.

Figure 11-10*a* shows the read timing waveforms for a Texas Instruments TMS44C256 DRAM which can be used for page mode access. For the first access in a row (page), the DRAM controller carries out a normal row address-$\overline{\text{RAS}}$, column address-$\overline{\text{CAS}}$ sequence of signals. If the next address the controller sends out is in the same row, an external comparator will send a signal to the DRAM controller. In response to this "same-row" signal, the DRAM controller will hold $\overline{\text{RAS}}$ low, send out just the column address to the A0–A8 inputs of the DRAMs, and pulse $\overline{\text{CAS}}$ low. As long as the microprocessor continues to access memory locations in the same page (row), the controller will simply hold $\overline{\text{RAS}}$ low, send out the column part of the addresses to the DRAMs, and pulse $\overline{\text{CAS}}$ low for each new column address. These accesses within a page are much faster because they require no row address and $\overline{\text{RAS}}$ time, and because they require no precharge time.

To determine if a memory access is within the same page, a device such as the SN74ALS6310 is connected to the address bus. This device holds the page part of the previous address in a register and compares it to the page part of the new address. If the two address parts are the same, the 6310 signals the DRAM controller to do a page mode access such as that shown in Figure 11-10*a*. If the previous page address and the current page address are different, the controller will do a normal $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ access.

Figure 11-10*b* shows the read timing waveforms for a Texas Instruments TMS44C257 DRAM which is designed for static column mode operation. During the first access in a row, the DRAM controller carries out a normal row address-$\overline{\text{RAS}}$, column address-$\overline{\text{CAS}}$ sequence of signals. If the next address the controller sends out is in the same row, an external comparator will signal the DRAM controller. In response to this "same-row" signal, the DRAM controller will hold $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ low and send out just the column address to the A0–A8 inputs of the DRAMs. As long as the microprocessor continues to access memory locations in the same page (row), the controller will simply hold $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ low and send out the column part of the addresses to the DRAMs. The static column mode is more difficult to implement than the page mode, but it is faster than the page mode because it does not require $\overline{\text{CAS}}$ strobes and the associated setup and hold times.

In a high-speed microprocessor system, the static column decode technique can reduce the average number of wait states per memory access from 2 or 3 to perhaps 0.8. This is a considerable improvement, but it is not as much of an improvement as can be gained by using a cache system.

Cache Mode DRAM Systems

INTRODUCTION

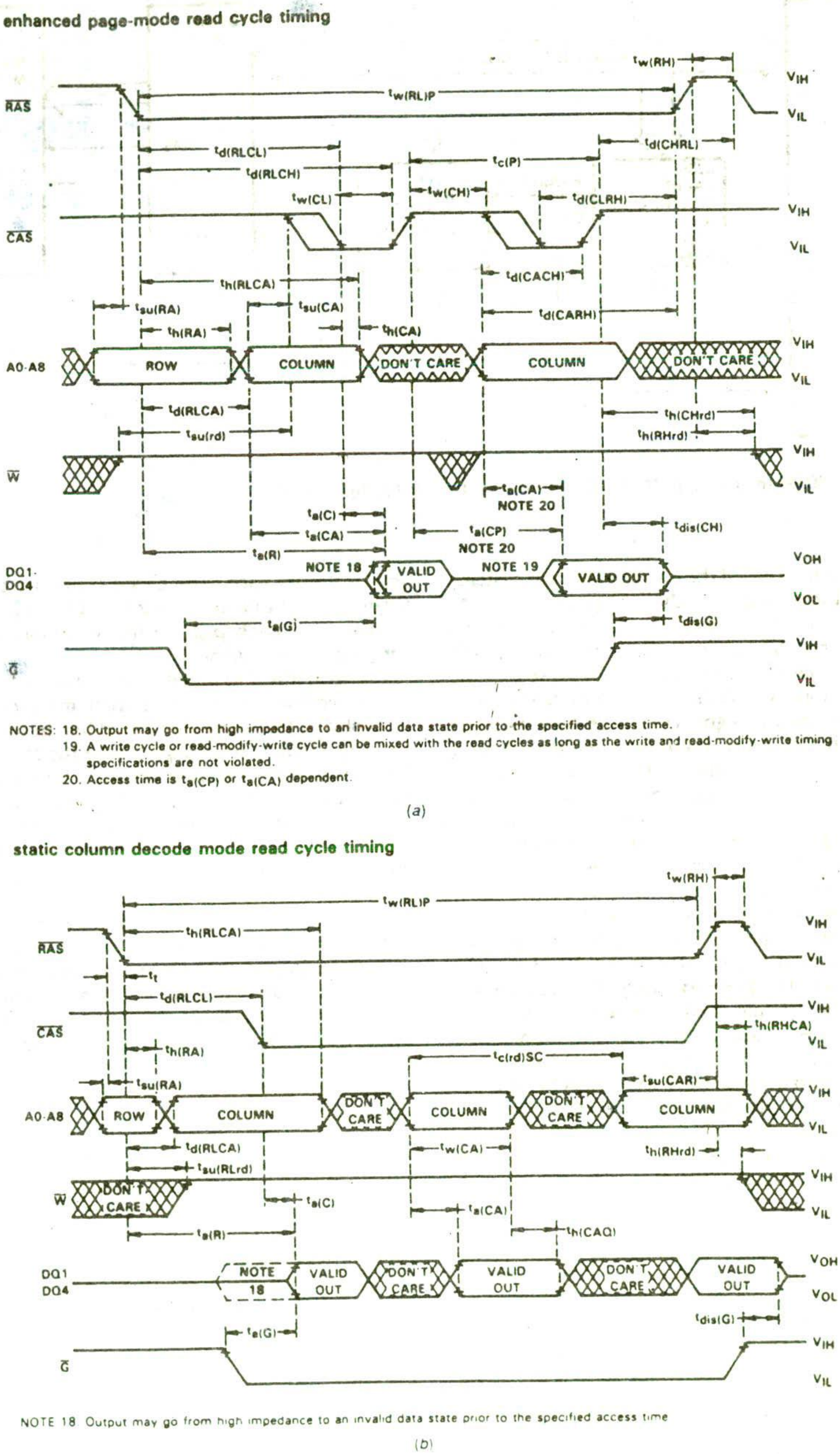
Traditionally the term *cache*, which is pronounced "cash," refers to a hiding place where you put provisions for future use. As we describe how a cache memory system is implemented in a microcomputer, perhaps you can see why the term is used here.

Figure 11-11, p. 360, shows in block diagram form how a simple cache memory system is implemented in an 80386 based microcomputer system. In Chapter 15 we discuss the details of the 80386 microprocessor, but for this discussion all you need to know is that the 80386 has a 32-bit data bus and a 32-bit address bus. A 32-bit address bus allows the 80386 to address up to 4 Gbytes of memory and a 32-bit data bus allows the 80386 to read or write 4 bytes in parallel.

The cache in a system such as this consists of perhaps 32 or 64 Kbytes of high-speed SRAM. The main memory consists of a few megabytes or more of slower but cheaper DRAM. The general principal of a cache system is that code and data sections currently being used are copied from the DRAM to the high-speed SRAM cache, where they can be accessed by the processor with no wait states. A cache system takes advantage of the fact that most microcomputer programs work with only small sections of code and data at a particular time. The fancy term for this is "locality of reference." Here's how the system works.

When the microprocessor outputs an address, the cache controller checks to see if the contents of that address have previously been transferred to the cache. If the addressed code or data word is present in the cache, the cache controller enables the cache memory to output the addressed word on the data bus. Since this access is to the fast SRAM, no wait states are required.

If the addressed word is not in the cache, the cache controller enables the DRAM controller. The DRAM controller then sends the address on to the main memory to get the data word. Since the DRAM main memory is slower, this access requires one or two wait states. However, when a word is read from main memory, it not only goes to the microprocessor, it is also written to the cache. If the processor needs to access this data word again, it can then read the data directly from the cache with no wait states. The percentage of accesses where the microprocessor finds the code or data word it



enhanced page-mode read cycle timing

NOTES: 18. Output may go from high impedance to an invalid data state prior to the specified access time.

19. A write cycle or read-modify-write cycle can be mixed with the read cycles as long as the write and read-modify-write timing specifications are not violated.

20. Access time is $t_{a(CP)}$ or $t_{a(CA)}$ dependent.

(*a*)

static column decode mode read cycle timing

NOTE 18. Output may go from high impedance to an invalid data state prior to the specified access time

(*b*)

FIGURE 11-10 TMS44C256 DRAM. (*a*) Page mode read-cycle operation. (*b*) Static column read-cycle operation. (*Texas Instruments Inc.*)

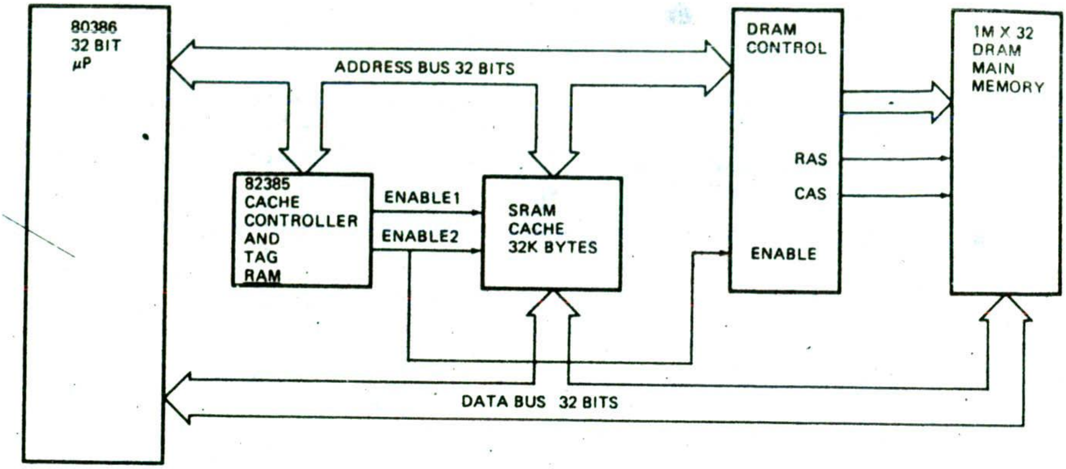


FIGURE 11-11 80386 microcomputer RAM memory system using high-speed SRAM cache.

needs in the cache is called the *hit rate*. Current systems have average hit rates greater than 90 percent.

For write to memory operations most cache systems use a *posted-write-through* method. If the cache controller determines that the addressed word is present in the cache, the controller will write the new word to the cache with no wait states and signal the 80386 that the write is complete. The controller will then write the data word to main memory. This write to the main memory is transparent to the main processor unless the main memory is still involved in a previous write operation.

To keep track of which main memory locations are currently present in the SRAM cache, the cache controller uses a *cache directory*. For the Intel 82385 cache controller shown in Figure 11-11, the cache directory RAM is contained in the controller. Each location in the cache is represented by an entry in the directory. The exact format for the directory entry depends on the particular cache scheme used. The three basic cache schemes are direct-mapped, two-way set associative, and fully associative. We don't have time here to do a detailed discussion of these three caching schemes, but we will give you an introduction to each so you will understand the terms if you see them in a computer magazine article or advertisement. We discuss cache systems further in Chapter 15.

A DIRECT MAPPED CACHE

Figure 11-12*a* shows a block diagram of how a direct-mapped 32 Kbyte cache can be implemented in an 80386 system with an 82385 controller. As we said before, an 80386 has a 32-bit address bus, so it can address 2^{32} bytes, or about 4 Gbytes, of memory. The 80386 also has a 32-bit data bus, so it can read up to 4 bytes at a time from memory. A group of four parallel bytes is commonly referred to as a line.

The cache memory for the 80386 system in Figure 11-12*a* is set up to hold 8K 4-byte lines or a total of 32 Kbytes. The 8K lines in the cache are organized as 1024 sets of 8 lines each. The cache controller treats the 4 Gbytes of main memory as 2^{17} or 131,072 pages of 32 Kbytes each. Each page in main memory then is the same size as the cache.

The term *direct mapped* here means that a particular numbered line from a page in main memory will always be copied to that same numbered line in the cache. For example, if line 1 from page 0 is in the cache, it will be stored in line 1 of the cache. If line 1 from page 131,070 is in the cache, it will be stored in line 1 of the cache.

The cache directory on the left of Figure 11-12*a* is used to keep track of which lines from the main memory currently have copies in the cache. As you can see, the directory contains a 26-bit entry for each set of 8 lines in the cache. The upper 17 bits of a directory entry are called a *tag*. The tag in a directory entry identifies the main memory page that a line or set of lines in the cache duplicates. Each directory entry also contains a *tag valid bit* and eight *line valid bits* (one for each line in the set). Here's how the 82385 uses this directory during a read operation.

When the 80386 sends out a 32-bit address to read a word from memory, address lines A15 through A31 represent a main memory page, address lines A5 through A14 identify the set containing a desired line, and address lines A2 through A4 identify the number of the line in the set containing the desired word. Figure 11-12*b* shows this in diagram form. The cache controller first uses address bits A5 through A14 to select the directory entry for the set that contains the addressed line. Then it compares the upper 17 bits of the address from the 80386 with the 17-bit tag stored in the directory entry. If the two are equal, the controller checks the tag valid bit to see if the tag is current. If the tag valid bit is set, the controller checks the line valid bit for the line addressed by address bits A2 through A4. If the tag matches and is valid and the line is valid, the line is in the cache. This is a cache *hit*. In this case the controller will apply address bits A2 through A14 to the cache

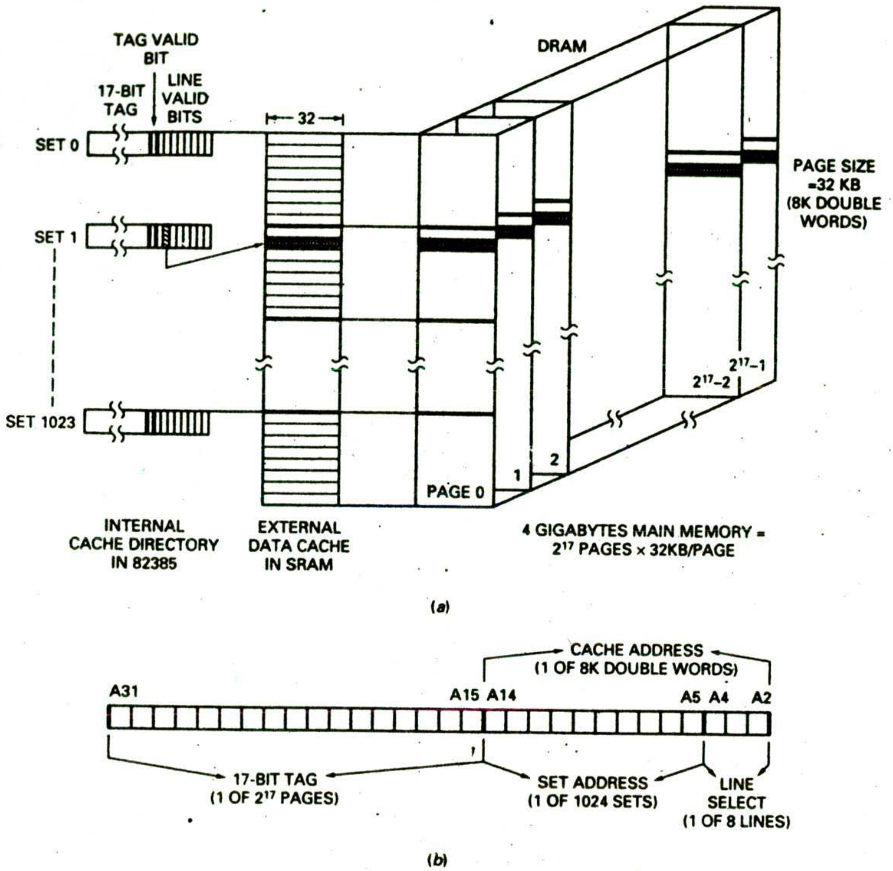

TAG VALID BIT
17-BIT TAG
LINE VALID BITS
32
SET 0
SET 1
SET 1023
DRAM
PAGE SIZE =32 KB (8K DOUBLE WORDS)
$2^{17}-1$
$2^{17}-2$
PAGE 0
1
2
INTERNAL CACHE DIRECTORY IN 82385
EXTERNAL DATA CACHE IN SRAM
4 GIGABYTES MAIN MEMORY = 2^{17} PAGES × 32KB/PAGE
(a)
CACHE ADDRESS (1 OF 8K DOUBLE WORDS)
A31
A15 A14
A5 A4
A2
17-BIT TAG (1 OF 2^{17} PAGES)
SET ADDRESS (1 OF 1024 SETS)
LINE SELECT (1 OF 8 LINES)
(b)

FIGURE 11-12 Cache organization for 32-Kbyte direct-mapped cache. (*a*) Block diagram. (*b*) Use of 32-bit address by 82385 cache controller.

memory and enable the cache memory to output the addressed word on the data bus.

If the upper 17 bits of the address from the 80386 are not the same as the tag in the directory, the tag bit is not valid, or the line bit for the addressed line is not valid, the read operation is a cache *miss*. In this case the 82385 will send the complete address from the 80386 along to the DRAM controller. The DRAM controller will cause the main memory to output the addressed line on the data bus. When this line appears on the data bus, the 82385 will enable the cache memory so that the line gets written to the cache as well as going to the 80386. The 82385 will also update the cache directory to indicate that this line is now in the cache. If this line or any part of it is needed again, it can be read directly from the cache.

When the 80386 writes a word to memory, the 82385 grabs the address and the data word then signals the 80386 that the transfer is complete. The controller then enables the main memory so that the word is written to the correct address in the main memory. If the data word is present in the cache, it is also written to the cache. This "posted write" process does not require any wait states unless the memory is still busy with a previous write.

A TWO-WAY SET ASSOCIATIVE CACHE SYSTEM

One difficulty with the direct-mapped cache approach is that if a program happens to use the same numbered line from two memory pages at the same time, it will be swapping the two lines back and forth between main memory and the cache as it executes. This swapping back and forth is called *thrashing*. A scheme which helps avoid thrashing is the *two-way set associative cache* approach shown in Figure 11-13*a*, p. 362. In this approach two separate caches and two separate cache directories are set up so that the same lines from two different pages can be cached at the same time. Each cache is half the size of the direct-mapped cache we discussed in the previous section, so the controller treats memory as 262,144 pages of 4096 lines each. To identify one of these 262,144 pages, the tag in each cache directory entry contains 18 bits. Each directory entry in this system also contains a tag valid bit, eight line valid bits, and a *least recently used bit* or LRU. Here's how this system works during a read operation.

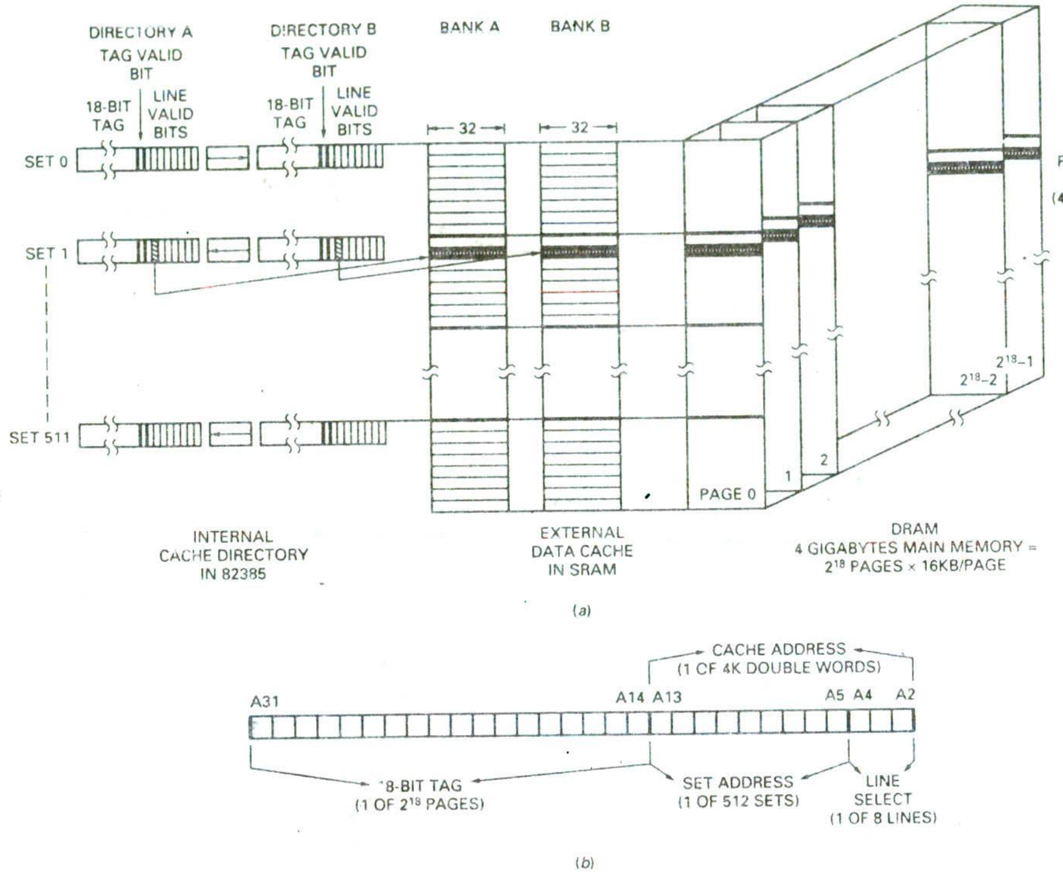

DIRECTORY A
TAG VALID BIT
18-BIT TAG
LINE VALID BITS
DIRECTORY B
TAG VALID BIT
18-BIT TAG
LINE VALID BITS
BANK A
BANK B
32
32
SET 0
SET 1
SET 511
PAGE SIZE =16 KB (4K DOUBLE WORDS)
$2^{18}-1$
$2^{18}-2$
2
1
PAGE 0
INTERNAL CACHE DIRECTORY IN 82385
EXTERNAL DATA CACHE IN SRAM
DRAM 4 GIGABYTES MAIN MEMORY = 2^{18} PAGES × 16KB/PAGE
(a)
CACHE ADDRESS (1 OF 4K DOUBLE WORDS)
A31
A14 A13
A5 A4
A2
18-BIT TAG (1 OF 2^{18} PAGES)
SET ADDRESS (1 OF 512 SETS)
LINE SELECT (1 OF 8 LINES)
(b)

FIGURE 11-13 Two-way set-associative cache for 32-bit address bus system. (*a*) Block diagram. (*b*) Use of 32-bit address by 82385 cache controller.

When the 80386 outputs an address, the 82385 controller uses address bits A5 through A13 to select the appropriate entry in each cache directory. It then compares the upper 18 bits of the address from the 80386 with the tag in each of the selected directory entries. If one of the tags matches, the controller checks the tag valid bit in that directory entry. The controller also checks the line valid bit for the line specified by address bits A2 through A4. If these bits are set, the controller outputs address bits A2 through A13 to the cache associated with that directory and enables the cache to output the desired word on the data bus.

If the addressed data word is found in cache A, the LRU bit in the directory entry is set to indicate that the A cache was most recently used. If the data word is found in the B cache, the LRU bit is set to indicate that the B cache was most recently used. This mechanism is used to determine which cache should be used to hold a new line read in from main memory. When a read operation produces a cache miss, the 82385 will send the address and control signals to the main memory to read a line containing the desired word. When this line comes down the data bus, the 82385 will write it to the least recently used cache and update the corresponding directory entry. If the controller finds that the tag for a read operation is correct but a line valid bit is invalid, it will read the line from main memory and write it in the cache whose directory contains the tag. This ensures that adjacent lines from a page in main memory end up in the same cache.

For a write operation this two-way set associative cache approach uses the same posted write-through method we described earlier. The controller always writes an output data word to the main memory, and if the word is present in one of the caches, it also updates the word in the cache.

Because of the two-tag RAMs, etc., this approach is somewhat more complex to implement, but it usually produces a better hit rate than a direct-mapped cache. The 25-MHz Compaq Deskpro 386/25 is an example of a system that uses an Intel 82385 cache controller and a two-way set associative cache to minimize wait states.

A FULLY ASSOCIATIVE CACHE SYSTEM

Still another type of cache that you may hear mentioned is the *fully associative* type. In this type a 4-byte block

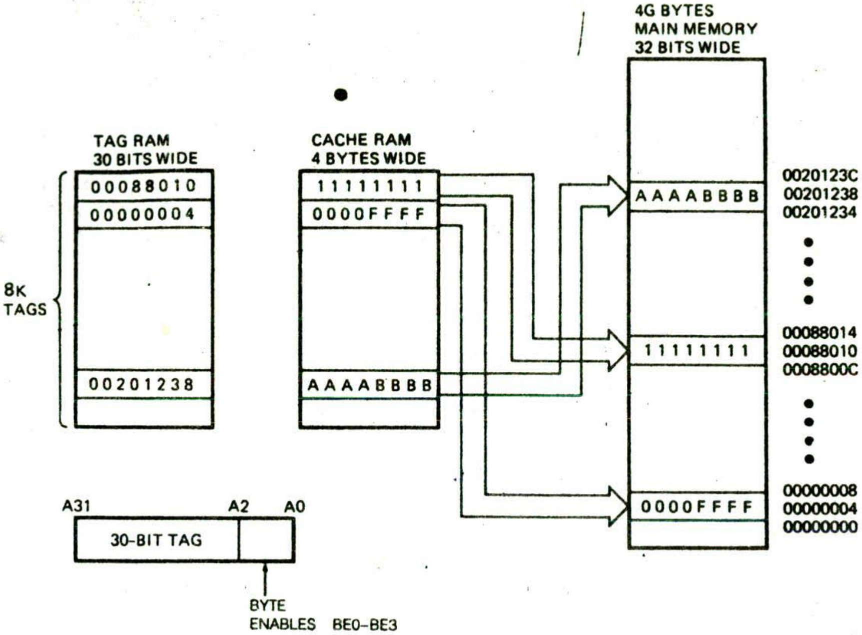


FIGURE 11-14. Fully associative 32-Kbyte cache for 32-bit address bus system.

or line from main memory can be written in any location in the cache. Figure 11-14 shows in block diagram form how this works.

The system has a 32-bit address bus, so it can address 4 Gbytes of memory. This corresponds to 1 Gbyte of 4-byte lines. Since 1 Gbyte is equal to 2^{30} bytes, a 30-bit tag is required to identify each block or line stored in the cache. Each entry in the directory then must contain 30 bits for the tag plus any additional bits used to keep track of how recently the line was used.

A fully associative cache has the advantage that it can hold the same numbered lines from several different pages at the same time. It has the disadvantage, however, that the upper 30 bits of each memory address sent out by the microprocessor must be compared with all of the tags in the directory to see if that line is present in the cache. This can be a time-consuming process. Also, when a fully associative cache is full, some algorithm must be used to determine which line to overwrite when a new line must be brought in from main memory. The most common algorithm replaces the least recently used line with the new line. The 82385, incidentally, is not designed to work with a fully associative cache system.

SUMMARY

The key point for you to remember about a cache is that by keeping the currently used code and data in a high-speed SRAM cache, the processor can use relatively inexpensive DRAM for its large main memory and still operate with few wait states. A cache controller device such as the 82385 automatically keeps the cache and the cache directory updated, so the process is essentially "invisible" to the microprocessor and to an executing program

Error Detecting and Correcting in DRAM Arrays

PARITY GENERATION/CHECKING

Data read from DRAMs is subject to two types of errors, *hard errors* and *soft errors*. Hard errors are caused by permanent device failures. These may be caused by a manufacturing defect or simply random breakdown in the chip. Soft errors are one-time errors caused by a noise pulse in the system or, in the case of dynamic RAMs, perhaps an alpha particle or some other radiation causing the charge to change on the tiny capacitor where a data bit is stored. As the size of a RAM array increases, the chance of a hard or a soft error increases sharply. This increases the chance that the entire system will fail. It seems unreasonable that one fleeting alpha particle may cause an entire system to fail. To prevent or at least reduce the chances of this kind of failure, we add circuitry which detects and in some cases corrects errors in the data read out from DRAMs. There are several ways to do this, depending on the amount of detection and correction needed.

The simplest method for detecting an error is with a parity bit. This is the method used in the IBM PC circuit shown in Figure 11-2. Note in this circuit that the DRAM memory bank is 9 bits wide. Eight of these bits are the data byte being stored, and the ninth bit is a parity bit which is used to detect errors in the stored data. A 74LS280 parity generator/checker circuit generates a parity bit for each byte and stores it in the ninth location as each byte is written to memory. When the 9 bits are read out, the overall parity is checked by the parity generator checker circuit. If the parity is not correct, an error signal is sent to the NMI logic to interrupt the processor. When you first turn on the power to an IBM

PC or warm boot it by pressing the Ctrl, Alt, and Del keys at the same time, one of the self-tests that it performs is to write byte patterns to all of the RAM locations and check if the byte read back and the parity of that byte are correct. If any error is found, an error message is displayed on the screen so you don't try to load and run programs in defective RAM.

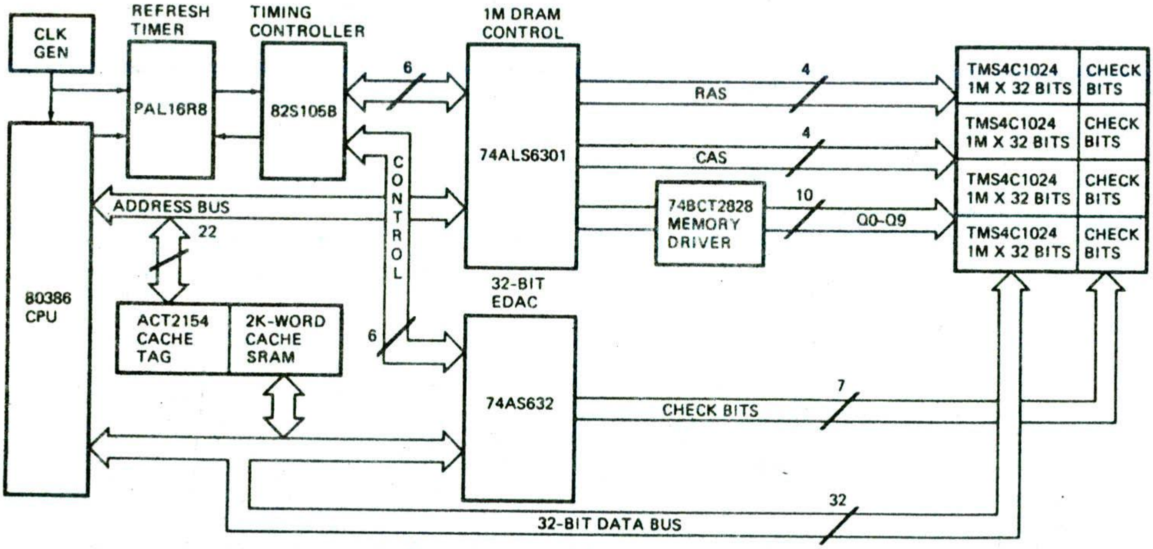
ERROR DETECTING AND CORRECTING CIRCUITS

One difficulty with a simple parity check is that two errors in a data word may cancel each other. A second problem with the simple parity method is that it does not tell you which bit in a word is wrong so that you can correct the error. More complex error detecting/correcting codes (ECCs), often called *Hamming codes* (after the man who did some of the original work in this area), permit you to detect multiple-bit errors in a word and to correct at least one bit error.

Figure 11-15*a* shows in block diagram form how a TI 74AS632 error detecting and correcting (EDAC) device can be connected in the data path between a 32-bit microprocessor and 16-Mbyte DRAM main memory. Note that the EDAC is connected in parallel with the DRAM refresh controller and in series with the SRAM cache. Here's how the EDAC device works.

When a data word is sent from the microprocessor to memory, it also goes to the EDAC. As the data word is read in by the EDAC, several *encoding* or *check* bits are generated and written in memory along with the data word. As shown in Figure 11-15*b*, the number of encoding bits, K, required is determined by the size of the data word, M, and the degree of detection/correction desired. The total number of bits required for a data word N is equal to M + K. For example, 5 encoding bits are required to detect and correct a single-bit error in a 16-bit data word, so a total of 21 bits have to be stored for each 16-bit word. To detect/correct a 1-bit error and detect 2 wrong bits in a 32-bit word requires 7 encoding bits, or a total of 39 bits. The encoding bits, incidentally, are not just tacked on to one end of the data word as a parity bit is. They are interspersed in the data word.

When the processor reads a data word from memory, the data word and the check bits from memory go to the EDAC. The EDAC calculates the check bits for the data word read out from memory and XORs these check bits with the check bits that were stored in memory with the data word. The result of this XOR operation is called a *syndrome word*. The syndrome word is decoded to determine if the data word has no errors, has a single-bit error, or has multiple-bit errors.



(*a*)

	SINGLE CORRECT/ SINGLE DETECT		SINGLE CORRECT/ DOUBLE DETECT	
K	≤ M ≤		≤ M ≤	
4	4	11	1	3
5	12	26	4	10
6	27	57	11	25
7	58	120	26	56
8	121	245	57	119

(*b*)

FIGURE 11-15 (*a*) Block diagram showing how error detecting and correcting circuitry is connected in a large DRAM system. (*Texas Instruments Inc.*) (*b*) Hamming-code data bits and encoding bits and number of encoding bits required for desired degree of detection/correction.

If the data word contains no errors, the 74AS632 EDAC will simply output the data word to the processor on the data bus. If the data word contains a single-bit error, the EDAC device uses the syndrome word to determine which bit is incorrect and simply inverts that bit to correct the bit. The EDAC then outputs the corrected data word to the processor on the data bus. If the data word contains multiple-bit errors, the EDAC device asserts a signal which is usually connected to an interrupt input on the processor. In the case of a multiple-bit error, the programmer must decide what action to take and write the appropriate interrupt-service procedure.

The 74AS632 EDAC in Figure 11-15*a* can also work with the 74ALS6301 DRAM controller to remove errors in stored data words during refresh operations as well as during normal read operations. This process is called *scrubbing*. Correcting errors during each refresh operation decreases the chance of multiple-bit errors accumulating between read operations.

For more information on DRAM error detecting/correcting, consult the data sheets for error detecting/correcting devices such as the Intel 8206, the Texas Instruments 74AS632, or the National DP8402A.

In the next section of this chapter we show you how a second processor can directly share the address, data, and control buses with the main processor in a microcomputer. Processors which share the local buses in this way are referred to as coprocessors. The example we use for this section is an Intel 8087 math coprocessor. As shown in Figure 11-2, the IBM PC and PC/XT have a socket for one of these devices.

A COPROCESSOR — THE 8087 MATH COPROCESSOR

Overview

Many microcomputer programs, such as those used for scientific research, engineering, business, and graphics, need to make mathematical calculations such as computing the square root of a number, the tangent of a number, or the log of a number. Another common need is to do arithmetic operations on very large and very small numbers. There are several ways to do all this.

One way is to write the number-crunching part of the program in a high-level language such as FORTRAN, compile this part of the program, and link in I/O modules written in assembly language. The difficulty with this approach is that programs written in high-level languages tend to run considerably slower than programs written in assembly language.

Another way is to write an assembly language program which uses the normal instruction set of the processor to do the arithmetic functions. Reference books which contain the algorithms for these are readily available. Our experience has shown that it is often time consuming to get from the algorithm to a working assembly language program.

Still another approach is to buy a library of floating-point arithmetic object modules from the manufacturer of the microprocessor you are working with or from an independent software house. In your program you just declare a procedure needed from the library as external, call the procedure as required, and link the library object code files for the procedures to the object code for your program. This approach spares you the labor of writing all the procedures.

In an application where you need a calculation to be done as quickly as possible, however, all the previous approaches have a problem. The architecture and instruction sets of general-purpose microprocessors such as the 8086 are not designed to do complex mathematical operations efficiently. Therefore, even highly optimized number-crunching programs run slowly on these general-purpose machines. To solve this problem, special processors with architectures and instruction sets optimized for number-crunching have been developed. An example of this type of number-crunching processor is the Intel 8087 math processor. An 8087 is used in parallel with the main microprocessor in a system, rather than serving as a main processor itself. Therefore, it is referred to as a *coprocessor*. The major principle here is that the main microprocessor, an 8088, for example, handles the general program execution and the 8087 coprocessor handles specialized math computations. An 8087 instruction may perform a given mathematical computation 100 times faster than the equivalent sequence of 8086 instructions.

An important point that we need to make about the 8087 is that it is an actual processor with its own, specialized instruction set. Instructions for the 8087 are written in a program as needed, interspersed with the 8088/8086 instructions. To you, the programmer, adding an 8087 to the system simply makes it appear that you have suddenly been given a whole new set of powerful math instructions to use in writing your programs. When your program is assembled, the opcodes for the 8087 instructions are put in memory right along with the codes for the 8086 or 8088 instructions. As the 8086 or 8088 fetches instruction bytes from memory and puts them in its queue, the 8087 also reads these instruction bytes and puts them in its internal queue. The 8087 decodes each instruction that comes into its queue. When it decodes an instruction from its queue and finds that it is an 8086 instruction, the 8087 simply treats the instruction as an NOP. Likewise, when the 8086 or 8088 decodes an instruction from its queue and finds that it is an 8087 instruction, the 8086 simply treats the instruction as an NOP or in some cases reads a data word from memory for the 8087. The point here is that each processor decodes all the instructions in the fetched instruction byte stream but executes only its own instructions. The first question that may occur to you is, How do the two processors recognize 8087 instructions? The answer is that all the 8087 instruction codes have 11011 as the most significant bits of their first code byte.

To start our discussion of the 8087 we will show you the data types, internal architecture, and programming of an 8087; then we will describe how an 8087 is connected and functions in a system. If you have an IBM PC or PC/XT type of computer, you can plug an

8087 chip in its auxiliary processor socket and run our example 8087 program or your own 8087 programs.

8087 Data Types

Figure 11-16 shows the formats for the different types of numbers that the 8087 is designed to work with. The three general types are binary integer, packed decimal, and real. We will discuss and show examples of each type individually.

BINARY INTEGERS

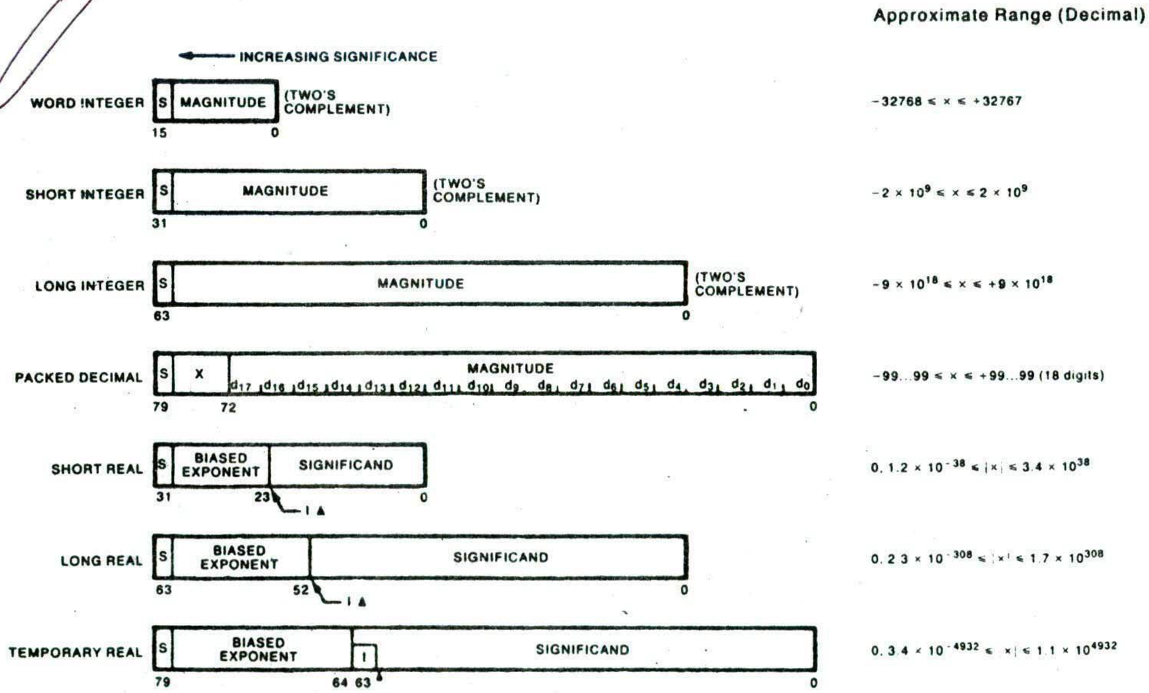
The first three formats in Figure 11-16 show different-length binary integer numbers. These all have the same basic format that we have been using to represent signed binary numbers throughout the rest of the book. The most significant bit is a sign bit which is 0 for positive numbers and 1 for negative numbers. The other 15 to 63 bits of the data word in these formats represent the magnitude of the number. If the number is negative, the magnitude of the number is represented in 2's complement form. Zero, remember, is considered a positive number in this format because it has a sign bit of 0. Note also in Figure 11-16 the range of values that can be represented by each of the three integer lengths. When you put numbers in this format in memory for the 8087 to access, you put the least significant byte in the lowest address.

PACKED DECIMAL NUMBERS

The second type of 8087 data format to look at in Figure 11-16 is the packed decimal. In this format a number is represented as a string of 18 BCD digits, packed two per byte. The most significant bit is a sign bit which is 0 for positive numbers and 1 for negative numbers. The bits indicated with an X are don't cares. This format is handy for working with financial programs. Using this format you can represent a dollar amount as large as $9,999,999,999,999,999.99, which is probably about what the national debt will be by the year 2000. Again, when you are putting numbers of this type in memory locations for the 8087 to access, the least significant byte goes in the lowest address.

REAL NUMBERS

Before we discuss the 8087 real-number formats, we need to talk a little about real numbers in general.



NOTES:
S = Sign bit (0 = positive, 1 = negative)
d_n = Decimal digit (two per byte)
X = Bits have no significance; 8087 ignores when loading, zeros when storing
▲ = Position of implicit binary point
I = Integer bit of significand; stored in temporary real, implicit in short and long real
Exponent Bias (normalized values):
Short Real: 127 (7FH)
Long Real: 1023 (3FFH)
Temporary Real: 16383 (3FFFH)

FIGURE 11-16 8087 data formats. *(Intel Corporation)*

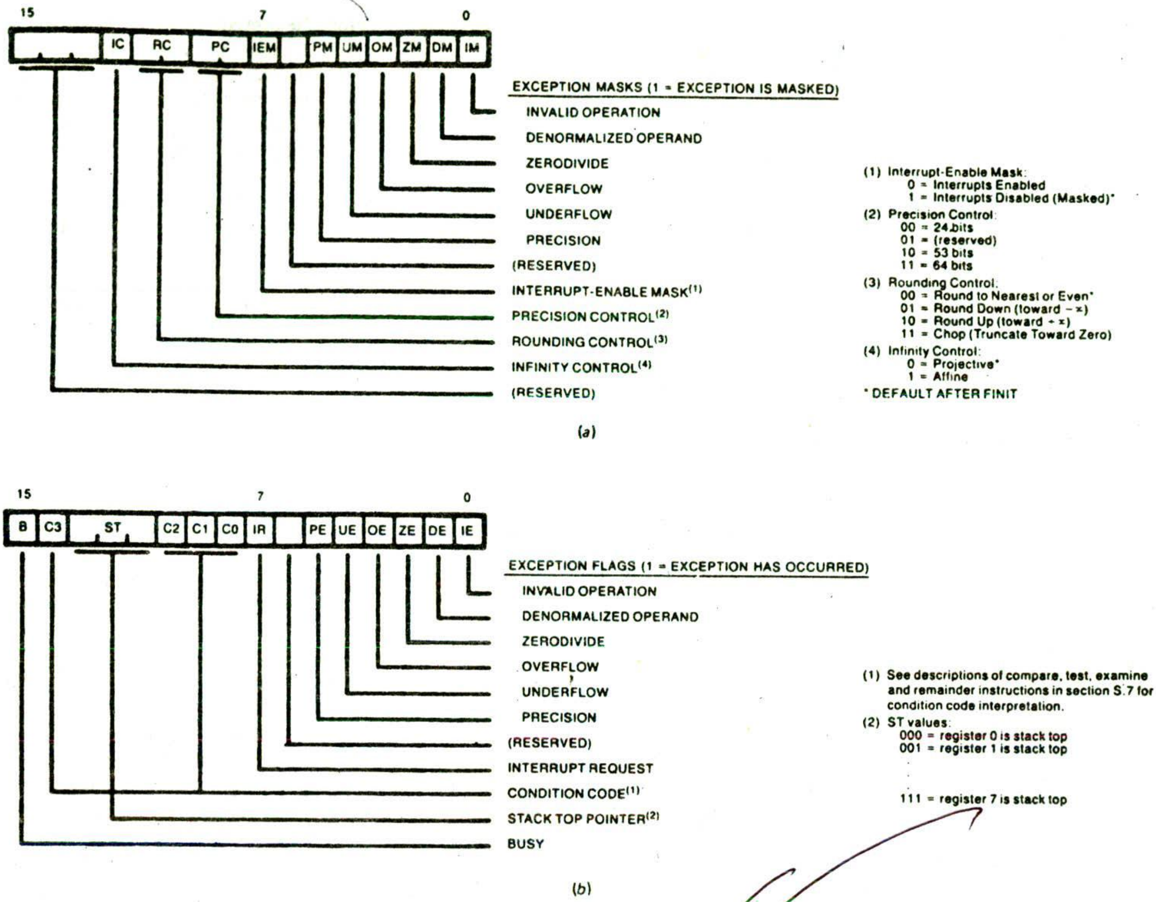


FIGURE 11-19 8087 control and status word formats. (*a*) Control. (*b*) Status. (*Intel Corporation*)

preceding paragraphs. To hold numbers being worked on, the 8087 has a register stack of eight 80-bit registers, labeled (0)–(7) in Figure 11-18. These registers are used as a last-in–first-out stack in the same way the 8086 uses a stack. The 8087 has a 3-bit stack pointer which holds the number of the register that is the current top-of-stack (TOS). When the 8087 is initialized, the 3-bit stack pointer in the 8087 is loaded with 000, so register 0 is then the TOS. When the 8087 reads in the first number that it is going to work on from memory, it converts the number to 80-bit temporary-real format if necessary. It then decrements the stack pointer to 111 and writes the temporary-real representation of the number in register number 111 (7). Figure 11-20*a*, p. 370, shows this in diagram form. As shown by the arrow in the figure, you can think of the stack as being wrapped around in a circle so that if you decrement 000 you get 111. From this diagram you can also see that if you push more than 8 numbers on the stack, they wrap around and write over previous numbers. After this write-to-stack operation, register 7 is now the TOS.

In the 8087 instructions the register that is currently the TOS is referred to as ST(0), or simply ST. The register just below this in the stack is referred to as ST(1). By the register "just below," we mean the register that the stack pointer would be pointing to if we popped one number off the stack. For the example in Figure 11-20*a*, register 000 would be ST(1) after the first push.

To help you understand this concept, Figure 11-20*b* shows another example. In this example we have pushed three numbers on the stack after initializing. Register 101 is now the TOS, so it is referred to as ST(0), or just ST. The preceding number pushed on the stack is in register 110, so it is referred to as ST(1). Likewise, the location below this in the stack is referred to as ST(2). If you draw a diagram such as that in Figure 11-20*b*, it is relatively easy to keep track of where everything is in the stack as instructions execute. In a program you can determine which register is currently the ST by simply transferring the status word to memory and checking the bits labeled ST in the status-word format in Figure 11-19*b*. Now let's have a look at the 8087 instruction set.

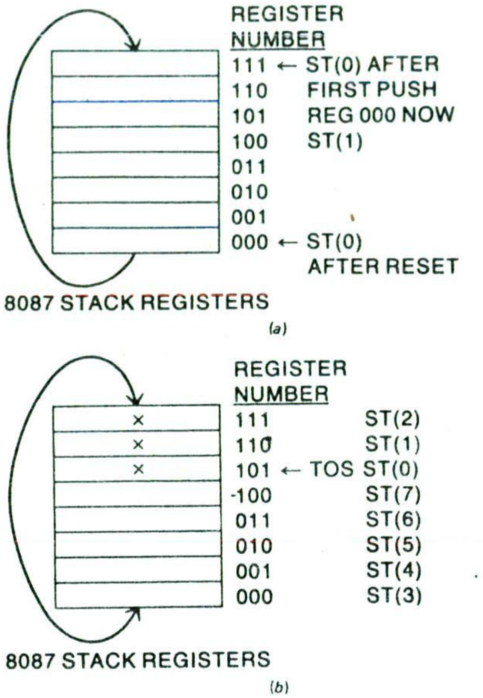


FIGURE 11-20 8087 stack operation. (*a*) Condition of stack after reset and one push. (*b*) Condition of stack after reset and three pushes.

8087 Instruction Set

8087 INSTRUCTION FORMATS

Before we work our way through the list of 8087 instructions, we will use one simple instruction to show you how 8087 instructions are written, how they operate, and how they are coded. The instruction we have chosen to use as an example here is the FADD instruction.

All the 8087 mnemonics start with an F, which stands for floating point, the form in which the 8087 works with numbers internally. If you look in the Intel data book, you will see this instruction represented as FADD // source/destination,source. This cryptic representation means that the instruction can be written in three different ways.

The // at the start indicates that the instruction can be written without any specified operands as simply FADD. In this case, when the 8087 executes the instruction, it will automatically add the number at the top of the stack, ST, to the number in the next location under it in the stack, ST(1). The 8087 stack pointer will be incremented by 1, so the register containing the result will be ST.

The word *source* by itself in the expression means that the instruction can be written as FADD source. The source specified here can be one of the stack elements or a memory location. For example, the instruction FADD ST(2) will add the number from two locations below ST to the number in ST and leave the result in ST. As another example, the instruction FADD CORRECTION_FACTOR will add a real number from the memory location named CORRECTION_FACTOR to the number in ST and leave the result in ST. The assembler will be able to determine whether the number in memory is a short-real, long-real, or temporary-real by the way that CORRECTION_FACTOR was declared. Short-reals, for example, are declared with the DD directive, long-reals with the DQ directive, and temporary-reals with the DT directive. If you want to add an integer number from memory to ST, you use an instruction such as FIADD CORRECTION_FACTOR. The I in the mnemonic tells the assembler to code the instruction so that the 8087 treats the number read in as an integer.

NOTE: The FIADD instruction only works for a source operand in memory.

The /destination,source in the representation of the FADD instruction means that you can write the instruction with both a specified source and a specified destination. The source can be one of the stack elements or a number from memory. The destination has to be one of the stack elements. The instruction FADD ST(2),ST(1), for example, will add the number one location down from ST to the number two locations down from ST and leave the result in ST(2). The instruction FADD ST(3),CORRECTION_FACTOR will add the real number from the memory location named CORRECTION_FACTOR to the contents of the ST(3) stack element.

Another form of the 8087 FADD instruction shown in the data book is FADDP. The P at the end of this mnemonic means POP. When the 8087 executes this form of the FADD instruction, it will increment the stack pointer by one after it does the add operation. This is referred to as "popping the stack." The instruction FADDP ST(1),ST(4), for example, will add the number at ST(4) to the number at ST(1) and put the result in ST(1). It will then pop the stack, or, in other words, increment the stack pointer so that what was ST(1) is now ST. This form of the instruction leaves the result at ST, where it can easily be transferred to memory. Now let's see how the different forms of this instruction are coded.

Coding 8087 Instructions

Common 8086 assemblers such as MASM and TASM accept 8087 mnemonics, and an assembler is the only practical way to produce codes for 8087 programs. However, to give you a feeling for how they are coded, we will show a few examples.

Figure 11-21 shows the coding templates for the 8087 FADD instructions as found in the Intel data book. Note that the figure shows coding for "8087" encoding and for "emulator" encoding. The 8087 encoding represents the codes required by an actual 8087 device. The emulator encoding represents the codes needed to call the FADD procedure from an available Intel library of 8086 procedures which perform the same functions as the 8087 instructions. The procedures in this library, written in 8086 code, run much slower, but they allow you

Type 1: Stack top and stack element

8087	10011011	11011 d 00	11000(i)
Emulator	11001101	00011 d 00	11000(i)

Type 2: Stack top and memory operand

8087	10011011	11011 m 00	mod 000 r/m
Emulator	11001101	00011 m 00	mod 000 r/m

m = 0 for short real operand; 1 for long real operand

Type 3: Pop stack

8087	10011011	11011110	11000(i)
Emulator	11001101	00011110	11000(i)

8087 Timing (clocks)	TYPICAL	RANGE
stack element and stack top	85	70-100
stack element, stack top + pop	90	75-105
short real memory and stack top	105+EA	90-120+EA
long real memory and stack top	110+EA	95-125+EA

FIGURE 11-21 8087 FADD coding templates. (*Intel Corporation*)

to test an 8087 program without having an actual 8087 in the system. We will concentrate here on the codes for the actual 8087 device.

First let's look at the coding for the FADD instruction with no specified operands. This instruction, remember, will add the contents of ST to the contents of ST(1), put the results in ST(1), and then pop the stack so that the result is at ST. The first byte of the instruction code, 10011011, is the code for the 8086 WAIT instruction. As we explain in detail later, this instruction code is put here to make the 8086 and 8087 wait until the 8087 has completed this instruction before starting the next one. The second byte shown is actually the first byte of the 8087 FADD instruction. The 5 most significant bits, 11011, identify this as an 8087 instruction. The lower 3 bits of the first code byte and the middle 3 bits of the second code byte are the opcode for the particular 8087 instruction. The bit labeled d at the start of these 6 bits is a 0 if the destination for an FADD ST(N),ST(N)-type instruction is ST. The d bit is a 1 if the destination stack element is one other than ST, as it is for the FADD instruction with no specified operands. For the FADD instruction with no specified operands, these 6 bits will be 100 000. The two most significant (MOD) bits in the second code byte are 1's because this form of the FADD instruction does not read a number from memory. The least significant 3 bits of the second instruction byte, represented by an i in the template, indicate which stack element other than ST is specified in the instruction. Since the simple FADD instruction uses ST(1) as a destination, 001 will be put in these bits. Putting all of this together for the FADD instruction with no specified source or destination gives 10011011 11011100 11000001 binary or 9BH DCH C1H as the code bytes.

For a little more practice with this, see if you can code the 8087 instruction FADD ST,ST(2). Most of the coding for this instruction is the same as that for the previous instruction. For this one, however, the d bit is a 0 because ST is the specified destination. Also, the R/M bits are 010, because the other register involved in the addition is ST(2). The answer is 9BD8C2H. Now let's try an example which uses memory as the source of an operand for FADD.

For an FADD instruction such as FADD CORRECTION_FACTOR, which brings in one operand from memory and adds it to ST, the memory address can be specified in any of the 24 ways shown in Figure 3-8. For the memory reference form of the FADD instruction, the MOD and R/M bits in the second code byte are used to specify the desired addressing mode. FADD CORRECTION_FACTOR represents direct addressing, so the MOD bits will be 00 and the R/M bits will be 110, as shown in Figure 3-8. Two additional code bytes will be used to put in the direct address, low byte first. Since we are not using any of the other stack elements other than ST for this instruction, we don't need the d bit to specify the other stack element. Instead, as shown in Figure 11-21, this bit is labeled m. A 0 in this bit is used to specify a short-real, and a 1 in this bit is used for a long-real. Assuming CORRECTION_FACTOR is declared as a long-real, the code bytes for our FADD CORRECTION_FACTOR instruction will then be 10011011 11011100 00000110 followed by the 2 bytes of the direct address.

Now that you have an overview of how 8087 instructions are written and coded, we briefly discuss each of the 8087 instructions.

8087 Instruction Descriptions

The 8087 instruction mnemonics all begin with the letter F, which stands for floating point and distinguishes the 8087 instructions from 8086 instructions. We have found that if we mentally remove the F as we read the mnemonic, it makes it easier to connect the mnemonic and the operation performed by the instruction. Here we briefly describe the operation of each of the 8087 instructions so that you can use some of them to write simple programs. As you read through these instructions the first time, don't try to absorb them all, or you probably won't remember any of them. Concentrate first on the instructions you need to get operands from memory into the 8087, simple arithmetic instructions, and the instructions you need to get results copied back from the 8087 to memory where you can use them. Then work your way through the example program in the next section. After that, read through the instructions again and pay special attention to the transcendental instructions which allow you to perform trigonometric and logarithmic operations.

The instructions are grouped here in six functional groups so that you can more easily find the instruction which performs a desired operation.

If the 8087 detects an error condition, usually called an *exception*, while it is executing an instruction, it will set the appropriate bit in its status register. After the instruction finishes executing, the status register contents can be transferred to memory with another 8087 instruction. You can then use 8086 instructions to check the status bits and decide what action to take if an error has occurred. Figure 11-19*b* shows the format of the 8087 status word. The lowest 6 bits are the exception status bits. These bits will all be 0's if no errors have occurred. In the instruction descriptions following, we use the first letter of each exception type to indicate the status bits affected by each instruction.

If you send the 8087 a control word which unmasks the exception interrupts, as shown in Figure 11-19*a*, the 8087 will also send out a hardware interrupt signal when an error occurs. This signal can be used to send the 8086 directly to an exception handling procedure.

DATA TRANSFER INSTRUCTIONS

Real Transfers

FLD source—Decrements the stack pointer by one and copies a real number from a stack element or memory location to the new ST. A short-real or long-real number from memory is automatically converted to temporary-real format by the 8087 before it is put in ST. Exceptions: I, D.

EXAMPLES:

```
FLD ST(3)              ; Copies ST(3) to ST
FLD LONG_REAL[BX]      ; Number from memory copied
                       ; to ST
```

FST destination—Copies ST to a specified stack position or to a specified memory location. If a number is transferred to a memory location, the number and its exponent will be rounded to fit in the destination memory location. Exceptions: I, O, U, P.

EXAMPLES:

```
FST ST(2)              ; Copy ST to ST(2), and
                       ; increment stack pointer
FST SHORT_REAL[BX]     ; Copy ST to memory
                       ; at SHORT_REAL[BX]
```

FSTP destination—Copies ST to a specified stack element or memory location and increments the stack pointer by 1 to point to the next element on the stack. This is a stack pop operation. It is identical to FST except for the effect on the stack pointer.

FXCH //destination—Exchanges the contents of ST with the contents of a specified stack element. If no destination is specified, then ST(1) is used. Exception: I.

EXAMPLE:

```
FXCH ST(5)   ; Swap ST and ST(5)
```

Integer Transfers

FILD source—Integer load. Convert integer number from memory to temporary-real format and push on 8087 stack. Exception: I.

EXAMPLE:

```
FILD DWORD PTR [BX]   ; Short integer from memory
                      ; at [BX]
```

FIST destination—Integer store. Convert number from ST to integer form and copy to memory. Exceptions: I, P.

EXAMPLE:

```
FIST LONG_INT    ; ST to memory locations
                 ; named LONG_INT
```

FISTP destination—Integer store and pop. Identical to FIST except that stack pointer is incremented after copy.

Packed Decimal Transfers

FBLD source—Packed decimal(BCD) load. Convert number from memory to temporary-real format and push on top of 8087 stack. Exception: I.

EXAMPLE:

```
FBLD MONEY_DUE    ; Ten byte BCD number from
                  ; memory to ST
```

FBSTP destination—BCD store in memory and pop 8087 stack. Pops temporary-real from stack, converts to 10-byte BCD, and writes result to memory. Exception: I.

EXAMPLE:

```
FBSTP TAX   ; ST converted to BCD, sent to memory
```

ARITHMETIC INSTRUCTIONS

Addition

FADD //source/destination, source—Add real from specified source to real at specified destination. Source can be stack element or memory location. Destination must be a stack element. If no source or destination is specified, then ST is added to ST(1) and the stack pointer is incremented so that the result of the addition is at ST. Exceptions: I, D, O, U, P.

EXAMPLES:

```
FADD ST(3), ST     ; Add ST to ST(3), result in ST(3)
FADD ST,ST(4)      ; Add ST(4) to ST, result in ST
FADD INTEREST      ; Real num from mem + ST
FADD               ; ST + ST(1), pop stack-result at ST
```

FADDP destination, source—Add ST to specified stack element and increment stack pointer by 1. Exceptions: I, D, O, U, P.

EXAMPLE:

```
FADDP ST(1)   ; Add ST(1) to ST. Increment stack
              ; pointer so ST(1) becomes ST
```

FIADD source—Add integer from memory to ST, result in ST. Exceptions: I, D, O, P.

EXAMPLE:

```
FIADD CARS_SOLD   ; Integer number from
                  ; memory + ST
```

Subtraction

FSUB //source/destination,source—Subtract the real number at the specified source from the real number at the specified destination and put the result in the specified destination. Exceptions: I, D, O, U, P.

EXAMPLES:

```
FSUB ST(2),ST   ; ST(2) becomes ST(2) - ST
FSUB CHARGE     ; ST becomes ST - real from memory
FSUB            ; ST becomes (ST(1) - ST)
```

FSUBP destination,source—Subtract ST from specified stack element and put result in specified stack element. Then increment stack pointer by 1. Exceptions: I, D, O, U, P.

EXAMPLES:

```
FSUBP ST(1)   ; ST(1) - ST. ST(1) becomes new ST.
```

FISUB source—Integer from memory subtracted from ST, result in ST. Exceptions: I, D, O, P.

EXAMPLE:

```
FISUB CARS_SOLD   ; ST becomes ST - integer
                  ; from memory
```

Reversed Subtraction

FSUBR //source/destination,source

FSUBRP //destination,source

FISUBR source—These instructions operate the same as the FSUB instructions described previously, except that these instructions subtract the contents of the specified destination from the contents of the specified source and put the difference in the specified destination. Normal FSUB instructions, remember, subtract source from destination.

Multiplication

FMUL //source/destination,source—Multiply real number from source by real number from specified destination and put result in specified stack element. See FADD instruction description for examples of specifying operands. Exceptions: I, D, O, U, P.

FMULP destination,source—Multiply real number from specified source by real number from specified destination, put result in specified stack element, and increment stack pointer by 1. See FADDP instruction for examples of how to specify operands for this instruction. With no specified operands FMULP multiplies ST(1) by ST and pops stack to leave result at ST. Exceptions: I, D, O, U, P.

FIMUL source—Multiply integer from memory times ST and put result in ST. Exceptions: I, D, O, P.

EXAMPLE:

```
FIMUL DWORD PTR [BX]
```

Division

FDIV //source/destination,source—Divide destination real by source real; result goes in destination. See FADD formats. Exceptions: I, D, Z, O, U, P.

FDIVP destination,source—Same as FDIV, but also increment stack pointer by 1 after DIV. See FADDP formats. Exceptions: I, D, Z, O, U, P.

FIDIV source—Divide ST by integer from memory, result in ST. Exceptions: I, D, Z, O, U, P.

Reversed Division

FDIVR //source/destination,source

FDIVP destination,source

FIDIVR source—These three instructions are identical in format to the FDIV, FDIVP, and FIDIV instructions, except that they divide the source operand by the destination operand and put the result in the destination.

Other Arithmetic Operations

FSQRT—Contents of ST are replaced with its square root. Exceptions: I, D, P.

EXAMPLE:

```
FSQRT
```

FSCALE—Scale the number in ST by adding an integer value in ST(1) to the exponent of the number in ST. Fast way of multiplying by integral powers of two. Exceptions: I, O, U.

FPREM—Partial remainder. The contents of ST(1) are subtracted from the contents of ST over and over again until the contents of ST are smaller than the contents of ST(1). FPREM can be used to reduce a large angle to less than π/4 so that the 8087 trig functions can be used on it. Exceptions: I, D, U.

EXAMPLE:

```
FPREM
```

FRNDINT—Round number in ST to an integer. The *round-control* (RC) bits in the control word determine how the number will be rounded. If the RC bits are set for down or chop, a number such as 205.73 will be rounded to 205. If the RC bits are set for up or nearest, 205.73 will be rounded to 206. Exceptions: I, P.

FXTRACT—Separates the exponent and the significand parts of a temporary-real number in ST. After the instruction executes, ST contains a temporary-real representation of the significand of the number and ST(1) contains a temporary-real representation of the exponent of the number. These two could then be written separately out to memory locations. Exception: I.

FABS—Number in ST is replaced by its absolute value. Instruction simply makes sign positive. Exception: I.

FCHS—Complements the sign of the number in ST. Exception: I.

COMPARE INSTRUCTIONS

The compare instructions with COM in their mnemonic compare contents of ST with contents of specified or default source. The source may be another stack element or real number in memory. These compare instructions set the condition code bits C3, C2, and C0 of the status word shown in Figure 11-19*b* as follows:

C3	C2	C0	
0	0	0	ST > source
0	0	0	ST < source
1	0	0	ST = source
1	1	1	numbers cannot be compared

You can transfer the status word to memory with the 8087 FSTSW instruction and then use 8086 instructions to determine the results of the comparison. Here are the different compares.

FCOM //source—Compares ST with real number in another stack element or memory. Exceptions: I, D.

EXAMPLES:

```
FCOM                         ; Compares ST with ST(1)
FCOM ST(3)                   ; Compares ST with ST(3)
FCOM MINIMUM_PAYMENT         ; Compares ST with real
                             ; from memory
```

FCOMP //source—Identical to FCOM except that the stack pointer is incremented by 1 after the compare operation. Old ST(1) becomes new ST.

FCOMPP—Compare ST with ST(1) and increment stack pointer by 2 after compare. This puts the new ST above the two numbers compared. Exceptions: I, D.

FICOM source—Compares ST to a short or long integer from memory. Exceptions: I, D.

EXAMPLE:

```
FICOM MAX_ALTITUDE
```

FICOMP source—Identical to FICOM except stack pointer is incremented by 1 after compare.

FTST—Compares ST with 0. Condition code bits C3, C2, and C0 in the status word are set as shown above if you assume the source in this case is 0. Exceptions: I, D.

FXAM—Tests ST to see if it is 0, infinity, unnormalized, or empty. Sets bits C3, C2, C1, and C0 to indicate result. See Intel data book for coding. Exceptions: None.

TRANSCENDENTAL (TRIGONOMETRIC AND EXPONENTIAL) INSTRUCTIONS

FPTAN—Computes the values for a ratio of *Y/X* for an angle in ST. The angle must be expressed in radians, and the angle must be in the range of $0 < \text{angle} < \pi/4$.

> NOTE: FPTAN does not work correctly for angles of exactly 0 and π/4. You can convert an angle from degrees to radians by dividing it by 57.295779. An angle greater than π/4 can be brought into range with the 8087 FPREM instruction. The *Y* value replaces the angle on the stack, and the *X* value is pushed on the stack to become the new ST. The values for *X* and *Y* are created separately so you can use them to calculate other trig functions for the given angle. Exceptions: I, P.

FPATAN—Computes the angle whose tangent is *Y/X*. The *X* value must be in ST, and the *Y* value must be in ST(1). Also, *X* and *Y* must satisfy the inequality $0 < Y < X < \infty$. The resulting angle expressed in radians replaces *Y* in the stack. After the operation the stack pointer is incremented so the result is then ST. Exceptions: U, P.

F2XM1—Computes the function $Y = 2^X - 1$ for an *X* value in ST. The result, *Y*, replaces *X* in ST. *X* must be in the range $0 \le X \le 0.5$. To produce 2^X, you can simply add 1 to the result from this instruction. Using some common equalities, you can produce values often needed in engineering and scientific calculations.

EXAMPLES:

$$10^X = 2^{X(\mathrm{LOG}_2 10)}$$
$$e^X = 2^{X(\mathrm{LOG}_2 e)}$$
$$Y^X = 2^{X(\mathrm{LOG}_2 Y)}$$

FYL2X—Calculates *Y* times the LOG to the base 2 of *X* or $Y(\mathrm{LOG}_2X)$. *X* must be in the range of $0 < X < \infty$ and *Y* must be in the range $-\infty < Y < +\infty$. *X* must initially be in ST and *Y* must be in ST(1). The result replaces *Y* and then the stack is popped so that the result is then at ST. This instruction can be used to compute the log of a number in any base, n, using the identity $LOG_nX = LOG_n2(LOG_2X)$. For a given n, LOG_n2 is a constant which can easily be calculated and used as the Y value for the instruction. Exceptions: P.

FYL2XP1—Computes the function Y times the LOG to the base 2 of $(X + 1)$ or $Y(LOG_2(X + 1))$. This instruction is almost identical to FYL2X except that it gives more accurate results when computing the LOG of a number very close to 1. Consult the Intel manual for further detail.

INSTRUCTIONS WHICH LOAD CONSTANTS

The following instructions simply push the indicated constant onto the stack. Having these commonly used constants available reduces programming effort.

FLDZ—Push 0.0 on stack.

FLD1—Push +1.0 on stack.

FLDPI—Push the value of π on stack.

FLD2T—Push LOG of 10 to the base 2 on stack (LOG_210).

FLDL2E—Push LOG of e to the base 2 on stack (LOG_2e).

FLDLG2—Push LOG of 2 to the base 10 on stack ($LOG_{10}2$).

FLDLN2—Push LOG of 2 to the base e on stack (LOG_e2).

PROCESSOR CONTROL INSTRUCTIONS

These instructions do not perform computations. They are used to do tasks such as initializing the 8087, enabling interrupts, writing the status word to memory, etc.

Instruction mnemonics with an N as the second character have the same function as those without the N, but they put an NOP in front of the instruction instead of putting a WAIT instruction there.

FINIT/FNINT—Initializes 8087. Disables interrupt output, sets stack pointer to register 7, sets default status.

FDISI/FNDISI—Disables the 8087 interrupt output pin so that it cannot cause an interrupt when an exception (error) occurs.

FENI/FNENI—Enables 8087 interrupt output so it can cause an interrupt when an exception occurs.

FLDCW source—Loads a status word from a named memory location into the 8087 status register. This instruction should be preceded by the FCLEX instruction to prevent a possible exception response if an exception bit in the status word is set.

FSTCW/FNSTCW destination—Copies the 8087 control word to a named memory location where you can determine its current value with 8086 instructions.

FSTSW/FNSTSW destination—Copies the 8087 status word to a named memory location. You can check various status bits with 8086 instructions and base further action on the state of these bits.

FCLEX/FNCLEX—Clears all the 8087 exception flag bits in the status register. Unasserts BUSY and INT outputs.

FSAVE/FNSAVE destination—Copies the 8087 control word, status word, pointers, and entire register stack to a named, 94-byte area of memory. After copying all this, the FSAVE/FNSAVE instruction initializes the 8087 as if the FINIT/FNINIT instruction had been executed.

FRSTOR source—Copies a 94-byte named area of memory into the 8087 control register, status register, pointer registers, and stack registers.

FSTENV/FNSTENV destination—Copies the 8087 control register, status register, tag words, and exception pointers to a named series of memory locations. This instruction does not copy the 8087 register stack to memory as the FSAVE/FNSAVE instruction does.

FLDENV source—Loads the 8087 control register, status register, tag word, and exception pointers from a named area in memory.

FINCSTP—Increments the 8087 stack pointer by 1. If the stack pointer contains 111 and it is incremented, it will point to 000.

FDECSTP—Decrements the stack pointer by 1. If the stack pointer contains 000 and it is decremented, it will contain 111.

FFREE destination—Changes the tag for the specified destination register to empty. See the Intel manual for a discussion of the tag word. (You usually don't need to know about it.)

FNOP—Performs no operation. Actually copies ST to ST.

FWAIT—This instruction is actually an 8086 instruction which makes the 8086 wait until it receives a not-busy signal from the 8087 to its $\overline{\text{TEST}}$ pin. This is done to make sure that neither the 8086 nor the 8087 starts the next instruction before the preceding 8087 instruction is completed.

An 8087 Example Program — Pythagoras Revisited

As you may remember from back there somewhere in geometry, the Pythagorean theorem states that the hypotenuse (longest side) of a right triangle squared is equal to the square of one of the other sides plus the square of the remaining side. This is commonly written as $C^2 = A^2 + B^2$. For this example program we want to solve this for the hypotenuse C, so we take the square root of both sides of the equation to give $C = \sqrt{A^2 + B^2}$.

Figure 11-22, p. 376, shows a simple 8087 program you can use to compute the value of C for given values of A and B. We have shown the assembler listing for the program so you can see the actual codes that are generated for the 8087 instructions. Note, for example, that each of the codes for the 8087 instructions here starts with 9BH, the code for the WAIT instruction whose function we explained before.

At the start of the program we set aside some named memory locations to store the values of the three sides

```
                             ;8087 PROGRAM F11-22.ASM
                             ;ABSTRACT:  NUMERIC DATA PROCESSOR EXAMPLE PROGRAM
                             ;           This program calculates the hypotenuse of a right
                             ;           triangle, given SIDE A and SIDE B.

                             .8087       ; This line tells TASM that the program contains 8087 instructions

                             NAME   PYTHAG

0000                         DATA SEGMENT      WORD    PUBLIC
0000  40400000                    SIDE_A       DD      3.0      ; Set aside space for Side A, short real
0004  40800000                    SIDE_B       DD      4.0      ; Set aside space for Side B, short real
0008  00000000                    HYPOTENUSE   DD      0        ; Set aside space for result, short real
                                                                ; 5.0 normalized = 40A00000
000C  0000                        CONTROL_WORD DW      0        ; Space for control word
000E  0000                        STATUS_WORD  DW      0        ; Space for status  word
0010                         DATA ENDS

0000                         CODE SEGMENT WORD PUBLIC
                                  ASSUME CS:CODE, DS:DATA
0000  B8 0000s               START: MOV   AX, DATA              ; Initialize data segment register
0003  8E D8                         MOV   DS, AX
0005  9B DB E3                      FINIT                       ; Initialize 8087
0008  C7 06 000Cr 03FF              MOV   CONTROL_WORD, 03FFH ; Put control word in memory so 8087 can access
                                                                ; it. Sets round to even & mask interrupts
000E  9B D9 2E 000Cr                FLDCW CONTROL_WORD          ; Load control to 8087
0013  9B D9 06 0000r                FLD   SIDE_A                ; Put value of SIDE_A on stack top
0018  9B D8 C8                      FMUL  ST, ST(0)             ; Square SIDE_A
001B  9B D9 06 0004r                FLD   SIDE_B                ; Put value of SIDE_B on stack top
0020  9B D8 C8                      FMUL  ST, ST(0)             ; Square SIDE_B
0023  9B D8 C1                      FADD  ST, ST(1)             ; (AxA + BxB), result at top of stack
0026  9B D9 FA                      FSQRT                       ; Take square root of ST, result in ST
0029  9B DD 3E 000Er                FSTSW STATUS_WORD           ; Copy status word to mem so 8086 can access it
002E  A1 000Er                      MOV   AX, STATUS_WORD       ; Bring status to AX to check for errors
0031  24 BF                         AND   AL, 0BFH              ; Mask unneeded bit
0033  75 05                         JNZ   STOP                  ; Handle error if found
0035  9B D9 1E 0008r                FSTP  HYPOTENUSE            ; No error, copy result from 8087 to memory
003A  90                     STOP:  NOP
003B                         CODE   ENDS
                                    END  START
```

FIGURE 11-22 8087 program to compute the hypotenuse of a right triangle.

of our triangle, the control word we want to send the 8087, and the status word we will read from the 8087 to check for error conditions. Remember, the only way you can pass numbers to and from the 8087 is by using 8087 instructions to read the numbers from memory locations or write the numbers to memory locations. In this section of the example program the statement SIDE_A DD 3.0 tells the assembler to set aside two words in memory to store the value of one of the sides of the triangle. The decimal point in 3.0 tells the assembler that this is a real number. The assembler then produces the short-real representation of 3.0 (40400000) and puts it in the reserved memory locations. Likewise, the statement SIDE_B DD 4.0 tells the assembler to set aside two word locations and put the short-real representation of 4.0 in them. The statement HYPOTENUSE DD 0 reserves a double-word space for the result of our computation. When the program is finished, these locations will contain 40A00000, the short-real representation for 5.0.

You would normally write the actual code section of this program as a procedure so that you could call it as needed. To make it simple here we have written it as a stand-alone program. We start by initializing the data segment register to point to our data in memory. We then initialize the 8087 with the FINIT instruction. The notations for the control word in Figure 11-19*a* show the default values for each part of the control word after FINIT executes. For most computations these values give the best results. However, just in case you might want to change some of these settings from their default values, we have included the instructions needed to send a new control word to the 8087. You first load the desired control word in a reserved memory location with the MOV CONTROL_WORD,03FFH instruction and then load this word into the 8087 with the FLDCW CONTROL_WORD instruction.

To perform the actual computation, we start at the inside of the equation and work our way outward. FLD SIDE_A brings in the value of the first side and pushes it on the 8087 stack. FMUL ST,ST(0) multiplies ST by ST and puts the result in ST, so ST = A^2. Next we bring in SIDE_B, push it on the 8087 stack with the FLD SIDE_B instruction, and square it with the FMUL ST,ST(0) instruction. ST now contains B^2 and ST(1) now contains A^2. We add tnese together and leave the result in ST with the FADD instruction. FSQRT takes the square root of the contents of ST and leaves the results in ST. To see if the result is valid, we copy the 8087 status word to the memory location we reserved for it with the FSTSW STATUS_WORD instruction. We then use 8086 instructions to check the six exception status bits to see if anything went wrong in the square root computation. If there were no exceptions (errors), these status bits will all be 0's, and the program will copy the result from ST to the memory location named HYPOTENUSE using the FSTP HYPOTENUSE instruction. We used the POP form of this instruction so that after this instruction, the stack pointer is back at the same register as it was when we started. This makes it easier to keep track of which register is ST, if necessary.

For the case where our test found an error had occurred, we could have program execution go to an error handling routine instead of simply to the STOP label as we did for this simple example.

Now that you know how it is programmed, let's look at how an 8087 is connected in a system and how it works with an 8086 or 8088 as it executes programs.

8087 Circuit Connections and Cooperation

Figure 11-23, p. 378, shows the first sheet of the schematics for the 256K version of the IBM PC. We chose this schematic not only to show you how an 8087 is connected in a system with an 8088 microprocessor, but also to show you another way in which schematics for microcomputers are commonly drawn.

First in Figure 11-23, note the numbers along the left and right edges of the schematic. These numbers indicate the other sheet(s) that the signal goes to. This is an alternative approach to the zone coordinates used in the schematics in Figure 7-8. In the schematic here the zone coordinates are not needed because all the input signal lines are extended to the left edge of the schematic, and all of the output signal lines are run to the right edge of the schematic. If you see that an output signal goes to sheet 10, then it is a simple task to scan down the left edge of sheet 10 to find that signal. The wide crosshatched strips in Figure 11-23 represent the address, data, and control buses. From the pin descriptions for the major ICs, you know where these signals are produced. You can then scan along the bus to see where various signals get dropped off at other devices. On this type of schematic the buses are always expanded to individual lines where they enter or leave a schematic. Now let's look at how the 8087 and 8088 are connected.

First note that the MIN/$\overline{\text{MX}}$ pin of the 8088 is grounded, so the 8088 is operating in its maximum mode. Remember that in maximum mode the 8088 sends out encoded control signals on the status lines S2, S1, and S0 and the queue status lines QS1 and QS0 instead of generating the control signals directly. In a maximum-mode system an external controller such as the 8288 at the bottom of Figure 11-23 decodes these status signals to produce the control signals. These status signals also go to the 8087 so that the 8087 can track the bus activity of the 8088.

Next observe that the multiplexed address/data lines, AD0–AD7, also go directly from the 8088 to the 8087. The 8088, remember, has the same instruction set as the 8086, but it has only an 8-bit data bus, so all read and writes are byte operations. The upper address lines, A8–A19, also connect directly from the 8088 to the 8087. The 8087 receives the same clock and reset signals as the 8088.

Now look at Figure 11-23 to see if you can figure out that the request/grant signal, $\overline{\text{RQ}}/\overline{\text{GT0}}$, from the 8087 is connected to the request/grant pin, $\overline{\text{RQ}}/\overline{\text{GT1}}$, of the 8088. The way you figure this out from the schematic is to notice that the signal from the 8087 $\overline{\text{RQ}}/\overline{\text{GT0}}$ pin is labeled just $\overline{\text{RQ}}/\overline{\text{GT}}$ where it enters the crosshatched bus. Likewise, the label on the signal coming from the crosshatched bus to the $\overline{\text{RQ}}/\overline{\text{GT1}}$ pin of the 8088 is also just labeled $\overline{\text{RQ}}/\overline{\text{GT}}$. You know from the fact that the two lines have the same label they are connected together. For an 8086 or an 8088 operating in maximum mode, this bidirectional line is used for DMA request/acknowledge signals. The 8087 uses DMA to transfer data between memory and its internal registers.

When the 8086 or 8088 reads an 8087 instruction that needs data from memory or wants to send data to memory, the 8086 sends out the memory address coded in the instruction and sends out the appropriate memory-read or memory-write signals to transfer a word of data. In the case of a memory-read, the addressed word will be put on the data bus by the memory. The 8087 then simply reads in this word off the data bus. The 8086 or 8088 ignores the data word. If the 8087 needs only this one word of data, it can then go on and execute its instruction. However, many 8087 instructions need to read in or write out up to 80-bit words. In these cases the 8088 outputs the address of the first data word on the address bus and outputs the appropriate memory-read or memory-write control signal. The 8087 reads the data word put on the data bus by memory or writes a data word to memory on the data bus. The 8087 then grabs the 20-bit physical address that was output by the 8086 or 8088. To transfer additional words it needs to or from memory, the 8087 then takes over the buses from the 8086. To take over the bus the 8087 sends out a low-going pulse on its $\overline{\text{RQ}}/\overline{\text{GT0}}$ pin, as shown in Figure 11-23. The 8086 or 8088 responds to this by sending another low-going pulse back to the $\overline{\text{RQ}}/\overline{\text{GT0}}$ pin of the 8087 and by floating its buses. The 8087 then increments the address it grabbed during the first transfer and outputs the incremented address on the address bus. When the 8087 outputs a memory-read or memory-write signal, another data word will be transferred to or from the 8087. The 8087 continues the process until it has transferred all the data words required by the instruction to or from memory. When the 8087 is through using the buses for its data transfer, it sends another low-going pulse out on its $\overline{\text{RQ}}/\overline{\text{GT0}}$ pin to let the 8086 or 8088 know it can have the buses back again. The key point here, then, is that the 8087 coprocessor can take over the buses from the *host* or *bus master* processor to transfer data when it needs to by pulsing the $\overline{\text{RQ}}/\overline{\text{GT0}}$ input of the host processor.

Another important connection to observe in Figure 11-23 is that between the BUSY signal from the 8087 and the $\overline{\text{TEST}}$ input of the 8088. As we mentioned earlier in the chapter, this connection and the 8086 WAIT instruction are used to make sure the 8086 or 8088

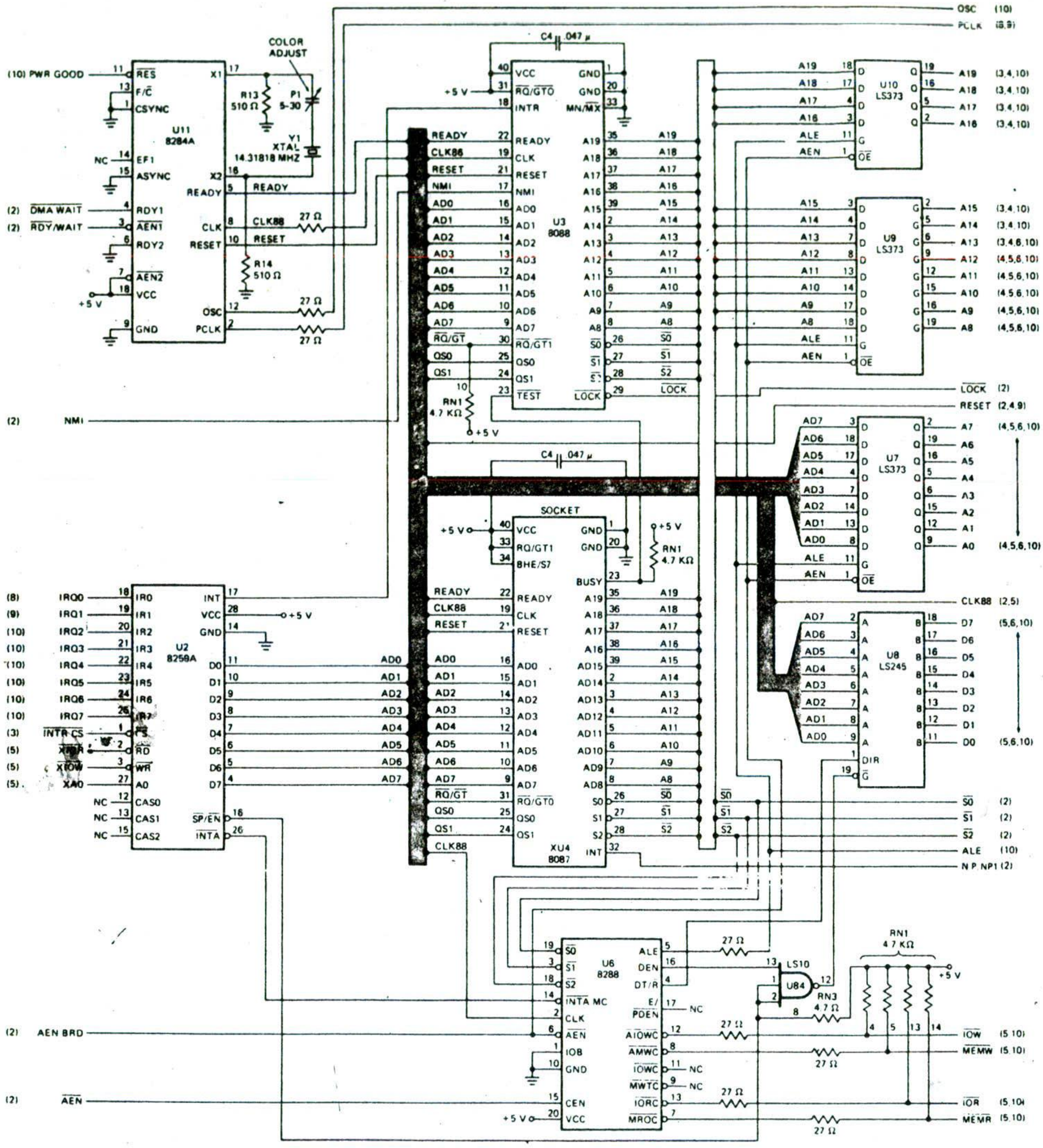


FIGURE 11-23 8088 and 8087 section of IBM PC schematic. (*IBM Corporation*)

host does not attempt to execute the next instruction before the 8087 has completed an instruction. There are two possible problem situations here.

One problem situation is the case where the 8086 needs the data produced by execution of an 8087 instruction to carry out its next instruction. In the instruction sequence in Figure 11-22, for example, the 8087 must complete the FSTSW STATUS_WORD instruction before the 8086 will have the data it needs to execute the MOV AX,STATUS_WORD instruction. Without some mechanism to make the 8086 wait until the 8087 completes the FSTSW instruction, the 8086 will go on and execute the MOV AX,STATUS_WORD instruction with erroneous data. This problem is solved by connecting the 8087 BUSY output to the TEST pin of the 8086 or 8088 and putting an 8086 WAIT instruction prefix.

9BH, before the 8087 FSTSW STATUS_WORD instruction in the program as shown in Figure 11-22. Here's how it works.

While the 8087 is executing an instruction, it asserts its BUSY pin high. The 8086 WAIT instruction prefix before the FSTSW STATUS_WORD instruction causes the 8087 to sit in a wait loop until its $\overline{\text{TEST}}$ pin is pulled low by the 8087 BUSY pin going low. The 8087 asserts its BUSY pin low when it completes the current instruction.

Another case where the host must be made to wait for the coprocessor is the case where a program has several 8087 instructions in sequence. The 8087 can obviously execute only one instruction at a time, so you have to make sure that the 8087 has completed one instruction before you allow the 8086 to fetch the next 8087 instruction from memory. Here again the BUSY-$\overline{\text{TEST}}$ connection and the FWAIT instruction prefix solve the problem.

As shown in the example program in Figure 11-22, an assembler will automatically insert the 8-bit code for the 8086 WAIT instruction, 10011011 binary (9BH), as the first byte of the code for each 8087 instruction. When the 8086 or 8088 fetches and decodes this code byte, it will enter on the internal loop and wait for the $\overline{\text{TEST}}$ input to go low before fetching and decoding the next 8087 instruction.

The final point we want to mention about the connections of the 8088 and 8087 in an IBM PC is that the INT output of the 8087 is connected to the nonmaskable interrupt (NMI) input of the 8088. This connection is made so that an error condition in the 8087 can interrupt the 8088 to let it know about the error condition. The signal from the 8087 INT output actually goes through some circuitry on sheet 2 of the schematics and returns to the input labeled NMI on the left edge of Figure 11-23. We do not have room here to show and explain all the circuitry on sheet 2. The main purposes of the circuitry between the INT output of the 8087 and the NMI input of the 8088 are to make sure that an NMI signal is not present during a reset, to make it possible to mask the NMI input, and to make it possible for other devices such as the parity checker to cause an NMI interrupt.

A couple of pins on the 8087 that we aren't concerned with in this system are the bus-high-enable (BHE) and request/grant1 (RQ/GT1) pins. When the 8087 is used with an 8086, the BHE pin is connected to the system BHE line to enable the upper bank of memory. The RQ/GT1 input is available so that another coprocessor can be connected in parallel with the 8087.

Next here we want to show you the tools and techniques currently used to design and test a microcomputer system such as the IBM PC shown in Figure 11-2.

COMPUTER-BASED DESIGN AND DEVELOPMENT TOOLS

In more and more companies the entire design, prototyping, manufacturing, and testing processes for an electronic product such as a microcomputer are being done with the help of a series of computer programs. The term *computer-aided engineering* (CAE) has been used in the past to describe the use of these tools, but now we commonly use the term *electronic design automation* or EDA instead of the more general term, CAE. The EDA term gives a better indication of the extent to which the currently available tools automate much of the design process. The following sections describe how a new microcomputer is designed and developed using design automation tools.

The Design Review Committee and Design Overview

The most important step in the design of any system is to think very carefully about what you want the system to do. In most companies a new product is now defined by a team consisting of design engineers, marketing/sales representatives, mechanical engineers, and production engineers. This team approach is necessary so that the product can be designed using current technology, manufactured and tested with minimal problems, and marketed successfully.

Once the specifications for the new product are agreed upon, the design engineers then think about how the circuit for it can be implemented. The next step in the design process is to partition the overall instrument design into major functional blocks or modules. Each module can then be individually designed and tested or even assigned to different designers.

Initial Design and Schematic Generation

The next step in the design process is to analyze each functional block to determine how it can best be implemented. After working out the basic design of each module, the design engineers draw a schematic for each module. In the old days (until about 5 years ago) we drew schematics on a large sheet of grid paper with a mechanical pencil and a plastic template. If we decided that a section of circuitry did not fit at a particular point on a schematic, we erased the block of circuitry with our electric eraser and started over again. The process was very time consuming and tedious. Just the remembrance of it makes me tired.

Now we use a *schematic capture* program to draw schematics on an engineering workstation such as the Apollo DN4500 shown in Figure 11-24, p. 380. Using a computer to draw schematics has the same advantage over hand drawing that using a word processor has over using a standard mechanical typewriter. Since the schematic is drawn on the computer screen, you don't have to erase anything on paper. You can move symbols around on the screen with a mouse, change connecting wires, add or delete symbols, and print out the result on a printer or plotter when the schematic looks just the way you want it to. If you change your mind about some part of the schematic, you can just edit the drawing on the screen and do a new printout.

A further advantage of the computer-aided drafting approach is that you usually don't even have to draw the symbols! Most schematic capture programs have

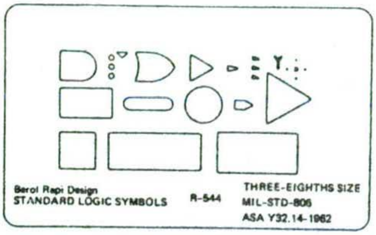


FIGURE 11-24 Apollo DN4500 workstation. *(Apollo Computers—HP Inc.)*

large library files containing common device symbols, complete with pin numbers and electrical characteristics. All you have to do when you want to put a particular IC on a schematic is to pull the symbol for it from the library file. Once you have the IC symbols for a circuit on the screen, you can use a mouse to draw the connecting wires between them and then add junctions, connectors, labels, etc., to complete the drawing.

Schematic capture programs are available for most computers. The Ideaware programs from Mentor Graphics run on Hewlett-Packard/Apollo engineering workstations such as the one shown Figure 11-24. We used a workstation such as this and the Mentor Graphics Neted schematic capture program to draw the basic microcomputer system in Figure 11-25. Workstations such as this are used for designing large, complex digital systems or ICs. For small projects we often use an IBM PC/AT or a Macintosh-type computer. Schematic capture programs for IBM PC-type computers include CapFast from Phase Three Logic, Draft from OrCAD Systems Corporation, Schema II+ from Omation, Inc., and EE Designer II from Visionics. Schematic capture programs available for the Macintosh include Schematic from Douglas Electronics Inc. and LogicWorks from Capilano Computing Systems Ltd.

When the schematic design file is completed, it is processed by a program called a *design rule checker* or *DRC*, which checks that there are no duplicate symbols, overlapped lines, or dangling lines. This step is similar to checking a text file with a spelling checker program.

After the schematic design file passes the DRC check, it is processed by a program called an *electrical rule checker* or *ERC*, which checks for wiring errors such as two outputs connected together or an output connected to V_{CC}.

When the schematic design file for a module passes the ERC test, a *netlist* program produces a netlist or wiring list for the design. A netlist is a file which lists all devices in the design and all the connections between devices.

Prototyping the Circuit — Simulation

After the design is polished, the next step is to prototype or "breadboard" the circuit design to make sure the logic and the timing in the circuit are correct. In the past this prototyping was usually done by soldering or wire-wrapping the circuit on a prototype board of some type. More and more we now use "software breadboarding" to test the operation of circuits of ICs. To do this we use a program called a *simulator*.

The simulator uses software *models* of the devices in the design to determine the response that the circuit will make to specified input signals. One big advantage of simulation or software breadboarding is that you don't have to order parts and wait for them to come in before you can test the operation of your design. Another big advantage of simulation is that you can change the design and resimulate the circuit in a matter of minutes to hours instead of waiting days for new parts to come in so you can modify a physical prototype and test it. Apollo Computer Corporation reportedly used simulation to cut several months from the prototype debug time for an engineering workstation such as the one shown in Figure 11-24.

Another advantage of simulation over traditional breadboarding is that you can simulate the circuit operation with worst-case timing parameters for all devices. This often discovers marginal timing problems that might not show up in a physical prototype because you can't vary the timing parameters of physical parts. We remember a case where a timing problem did not show up in the wire-wrapped prototype but caused a 40 percent failure rate in the first production run of the instrument.

As we said before, a simulator program uses models of the devices in the circuit to determine the effect that specified input signals will have on the outputs of the circuit. Most models are just software descriptions of the characteristics of the devices. These descriptions are usually written in a high-level programming language such as Pascal or C. As a simple example, part of the model for a basic, 3-input AND gate might look something like the following.

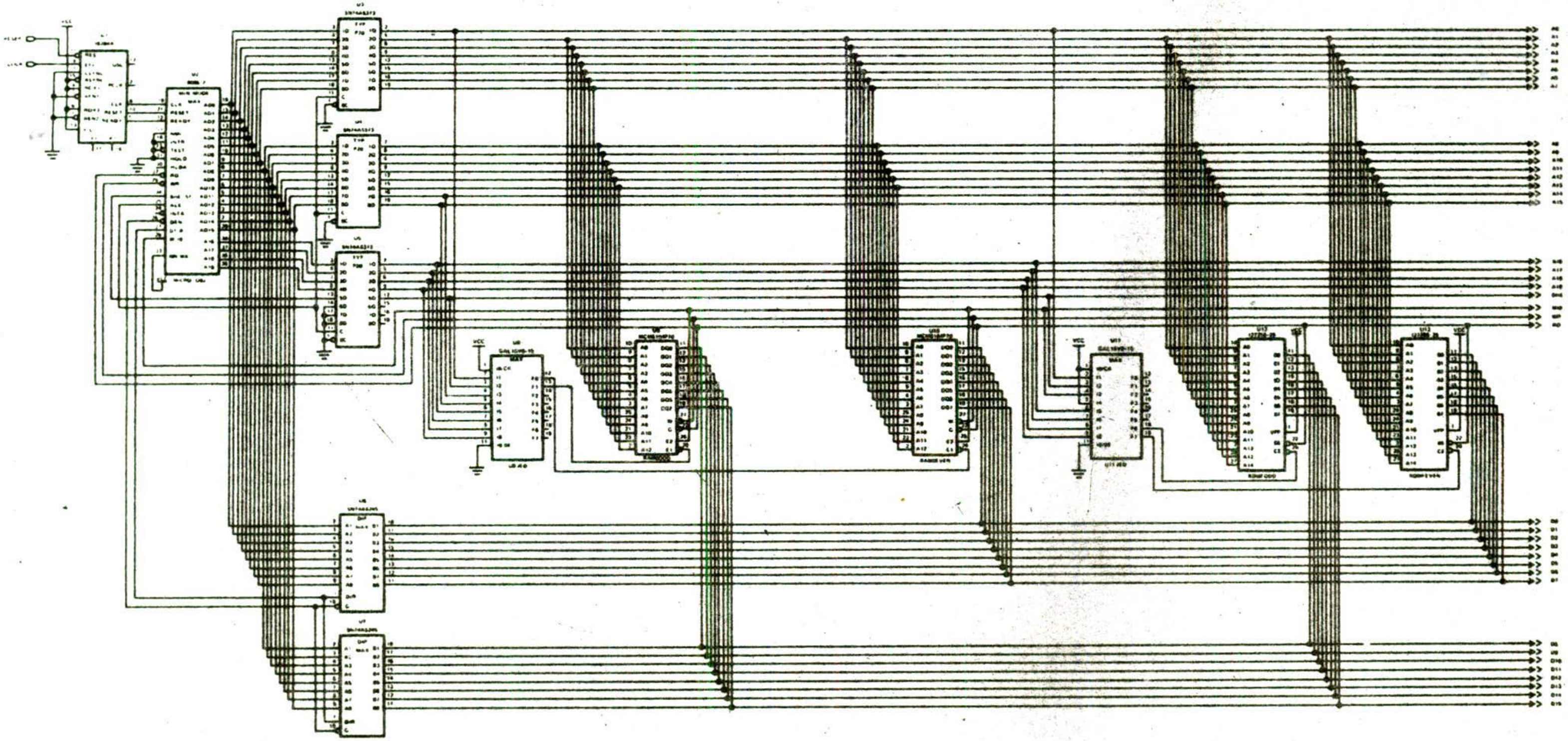


FIGURE 11-25 Schematic for simple 8086 microcomputer drawn with Mentor Graphic's Neted schematic capture program.

```
PROCEDURE ANDGATE   ;
  CONST        TPLH = 15;
               TPHL = 10;
 VAR IN1, IN2, IN3  :  INTEGER ;
     DELAY, OUT     :  INTEGER ;
  BEGIN
    IF (IN1 = 1) AND (IN2 = 1) AND (IN3 = 1)
       THEN BEGIN
              DELAY  : =  TPLH
              OUT    : =  1;
              END
                ELSE BEGIN
                       DELAY  : =  TPHL
                       OUT    : =  0
                       END
 END;
```

This model is very primitive, but it should give you the idea. The constants represent the characteristics of the specific device being simulated (TPLH and TPHL). The variables represent the input logic levels (IN1–IN3), the output logic level (OUT), and the time between a change on the input and the corresponding change on the output (DELAY). Some simulators refer to these characteristics as *properties*. The schematic symbol is really part of the model for a device, so when you draw a schematic with a schematic capture program, you are actually creating a design file which contains the logical and timing characteristics of each device as well as the schematic symbols and connections.

When you set up the simulator to do a simulation run, you specify the signals you want applied to the inputs at a particular time, just as you connect signal generators to the inputs of a physical circuit. The simulator uses the model to determine the effects that the specified input signals will have on the output and schedules the output to change appropriately after the delay time for that device. As you can see, the model for the 3-input AND gate device tells the simulator program that if the input signals become all 1's, the output should be scheduled to change to a 1 after 15 ns. If the inputs change to a case where they are not all 1's, the output should be scheduled to change to a 0 after 10 ns.

The smallest increment of time used by a simulator is called its *time step*. You can think of the time step as the time resolution of the simulator. For simulating TTL and CMOS circuits, simulators usually use a time step of 1 ns or 0.1 ns because the delay times for these devices are a few ns. An important point here is that the 0.1-ns time step is simulator time, not real time. The simulator may take 20 minutes to determine the effects that some input signal changes produce on the outputs of a complex circuit. The physical circuit would respond to the same input changes in a real time of just a few nanoseconds. The simulator essentially exercises the circuit "in slow motion" and generates an output which *represents*, or "simulates," the real-time operation of the circuit.

Now that you have an overview of how a simulator uses models, we need to talk briefly about some of the commonly used types of models. Three of these types are:

Gate-level models

Behavioral models

Hardware models

As you may remember from a basic logic course, any digital circuit can be implemented with just basic gates. We didn't bother to show you, but even a complex device such as an 8086 or 80386 microprocessor can be modeled at the basic gate level for simulation. The difficulty with using gate-level models for complex devices is that simulation using these models requires a very long time. The reason for this is that the simulator must evaluate the effects of each signal change on all the intermediate circuit points *(nodes)* in the device.

If the complex device is a standard part, we usually know that all the internal circuitry works correctly, so we don't need to resimulate at the gate level of detail. To speed up the simulation of circuits containing complex devices, we often use *behavioral models*. Behavioral models simply describe the effects that input signals will have on the output signals and the signal delays between inputs and outputs. A behavioral model of a D flip-flop, for example, will indicate that 20 ns after a positive clock edge, the logic level on the D input will be transferred to the Q output if neither the Preset nor the Clear input is asserted. Behavioral models also contain properties such as setup times, hold times, and minimum pulse widths so the simulator can check for violations of these times by the signals propagating through the circuit. Sophisticated behavioral models such as the "SmartModels" from Logic Automation Inc. give detailed error messages to pinpoint a timing problem instead of making you work your way through a logic-analyzer-type display to find the problem.

For simulating microprocessors there are two types of behavioral models available. One type is called a *hardware verification model*. This type model is essentially a "black box" which will, for example, produce the correctly timed address and control bus signals for a memory-read cycle when given the proper *processor control language (PCL)* file. Hardware verification models are easy to use for checking system timing because all they need is a simple PCL file as a stimulus. However, hardware verification models do not allow simulation of actual microprocessor instructions. If we need this level of simulation, we use *full functional models*, which do allow the execution of instructions. The disadvantages of full functional models are that they operate more slowly than hardware verification models and you have to develop a file containing the actual object codes for the microprocessor instructions you want to execute.

In cases where a behavioral model of a device is not available and it is not practical to write a model or in cases where the simulation must interfere with external circuitry at real time speeds, we use *hardware modeling*. In this approach the devices to be simulated are plugged into an external unit such as the Mentor Graphics Hardware Modeling System (HML) shown in Figure 11-26. When using a unit such as this, the simulator program sends stimulus signals to the external devices,

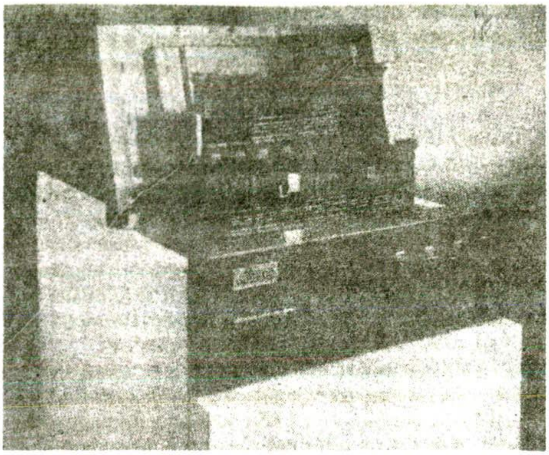


FIGURE 11-26 Mentor Graphics HML box.

reads back the responses of the external devices, and includes these responses in the simulation.

To develop complex systems such as the engineering workstation shown in Figure 11-24 we use *multilevel* simulators. An example of a multilevel simulator is Mentor Graphics Quicksim, which can simulate combinations of gate, behavioral, and hardware models. Quicksim runs on engineering workstations such as the one in Figure 11-24. Another useful but somewhat less powerful, multilevel simulator is SUSIE from Aldec Corp. SUSIE runs on PC-type computers and is available to schools at a generous discount. Multilevel simulators such as this even allow the JEDEC files for PALs to be included in the simulation.

To simulate analog circuits, you can use an analog circuit simulator such as PSPICE from Microsim Corporation or Accusim from Mentor Graphics. For circuits such as A/D converters, which have both analog and digital circuitry, you can use a *mixed-mode simulator* such as SABER from Analogy, Inc. or LSIM from Mentor Graphics.

A Microcomputer Simulation Example

We drew the schematic for the basic 8086 based microcomputer in Figure 11-25 using Mentor Graphics Neted and Logic Automation Smartmodels. As you can see in the figure, the circuit uses SN74AS373s as address latches and SN74AS245s as data bus buffers. The ROM in this systems consists of two 127256 EPROMs, one for the even bank and one for the odd bank. A Lattice GAL16V8 EPLD is used as an address decoder for the ROMs. The RAM in this basic system consists of two MCM6164 static RAM devices, one for the even bank and one for the odd bank. A second Lattice GAL16V8 EPLD is used as an address decoder for the RAMs. Offpage connectors go to a second sheet which contains the ports, timers, etc. For this example we are interested only in the basic microprocessor and memory section of the system.

The Logic Automation Smartmodel for the 8086 processor in Figure 11-25 is a hardware verification type. As we said above, this type model allows us to verify that the signal connections, address decoding, and timing of the system are correct. To refresh your memory as to what is involved in the timing of a system such as this, take another look at Figure 7-13.

As you can see in Figure 7-13*a*, the 8086 and memories essentially form a loop. To read a word from memory the 8086 sends out an address and control signals, and after some propagation delay the memory sends the data word back to the 8086. In order for the data word to be accepted by the 8086, it has to get back to the 8086 within a certain time period. In Figure 7-20 and the accompanying discussion, for example, we showed you how to determine if the address access time of a 2716 EPROM was fast enough for the device to work in a 4.9-MHz 8086 system.

When you use Smartmodels to simulate a system such as that in Figure 11-25, the simulator will automatically perform all the memory timing computations and give you an error message if it finds any timing violations. You can then redesign the circuit and resimulate until you get no error messages.

To simulate the circuit you have to give the simulator several types of information in addition to the basic netlist produced from the schematic. These additional parts are put in files which the simulator will read out as it needs them. The process is really quite simple. Here is a list of the parts you need.

1. A fusemap or JEDEC file for each of the GAL16V8 EPLD address decoders. These can be produced with a PAL programming tool such as ABEL from Data I/O. Figure 11-27*a*, p. 384, shows an ABEL source file for the U11 ROM decoder.

2. A memory image file for each of the memory devices. These are simple test files which essentially initialize the memory devices with known contents so you will know if data is read back correctly. Figure 11-27*b* shows a memory image file which will initialize the first 100H locations of a memory device with 88H.

3. A processor control (PCL) file which tells the simulator the bus operations you want the processor to perform. For Logic Automation Smartmodels this file is written in C. Figure 11-27*c* shows an example of a PCL file for our 8086 system. The instructions in the first block write bytes to a sequence of RAM locations starting at address 00000H. After it is written, each byte is read back. After the simulation is run the results from this part help us determine if the address decoding, control signals, and timing are correct for the RAM part of the circuit. The next block in Figure 11-27*c* reads data words from a series of ROM locations to verify the address decoding, control signals, and timing of the ROM section of the circuit. If we were also simulating programmable peripheral devices, we would include a section in the PCL file to initialize the devices and exercise their functions.

4. A stimulus file which tells the simulator what signals

```
module rompal
title '8086 SYSTEM ROM DECODER - DOUG HALL 1988'

    U11                        DEVICE 'P16V8S';
    A0,BHE,MIO                 PIN 2,3,4;
    A16,A17,A18,A19            PIN 6,7,8,9;
    ROMF_EVEN, ROMF_ODD        PIN 19,18;
    ROME_EVEN, ROME_ODD        PIN 17,16;

EQUATIONS

    !ROMF_EVEN = A19 & A18 & A17 &  A16 & !A0  & MIO;
    !ROMF_ODD  = A19 & A18 & A17 &  A16 & !BHE & MIO;
    !ROME_EVEN = A19 & A18 & A17 & !A16 & !A0  & MIO;
    !ROME_ODD  = A19 & A18 & A17 & !A16 & !BHE & MIO;
END rompal
```

(a)

```
0:100/88;
```

(b)

```
   #include <i8086min.cmd>
    int i,addr;
main( )
{
    trace_on( );
    set_trace_level(1);
    addr = 0x0000;

    for (i=0; i<=16; i++)
    {
    write(1,addr,i);
    read(1,addr);
    idle(5);
    addr++;
    }
    addr = 0xf0000;
       i = 0;
    for (i=0; i<=16; i++)
    {
    read(2,addr);
    addr++;
    addr++;
    }
}
```

(c)

```
CLOCK PERIOD 125
FORCE CLOCK 0 0    -R
FORCE CLOCK 1 62.5 -R
FORCE RESET 0 0
FORCE RESET 1 1000
```

(d)

FIGURE 11-27 Files required for simulating microcomputer circuit in Figure 11-25. (*a*) ABEL source file for PAL address decoder. (*b*) Memory image file. (*c*) Processor control file. (*d*) Simulator stimulus file.

to apply to the signal inputs of the system so that it runs through the operations in the PCL file. Figure 11-27*d* shows an example. The first three statements generate an 8-MHz clock for the external clock input of the 8284 clock generator. The next two statements generate a RESET signal. Note that the numbers such as 125, 62.5, and 1000 in these statements represent times in nanoseconds.

Once you have generated the necessary files, all you have to do is run the simulation. Figure 11-28 shows the screen messages produced as Quicksim is invoked and run on our microcomputer system design. As you can see, Quicksim first loads the required files in memory. When the Quicksim prompt appears, we execute the stimulus file in Figure 11-27*d* with DO MICRO.DO command. Then we run the simulator for 100,000 time units of 1 ns with the RUN 100000 command.

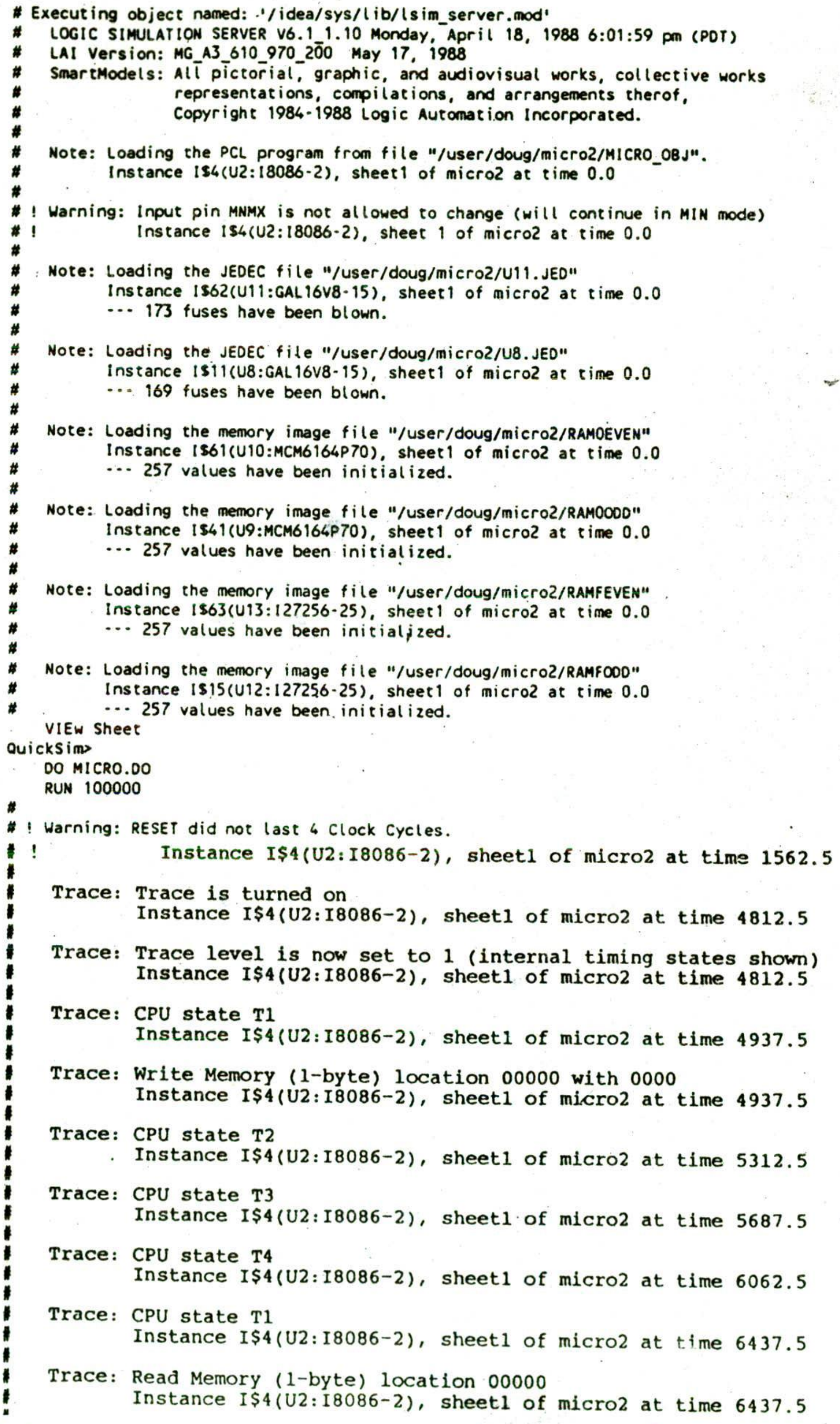
When the simulator started running, it immediately gave an error message indicating that we did not hold the RESET input of the 8086 low for the four clock periods required by the manufacturer's specifications. The problem here is that in our force file shown in Figure 11-27*d*, we generated an 8-MHz clock on the external clock input of the 8284 clock generator and held RESET low for 1000 ns, or eight of these clock cycles. The 8284, however, divides the external clock signal by 3 tc produce the clock signal actually applied to the processor. This means that in actuality our reset stimulus was holding the RESET input low for only a little more than two cycles of the clock applied to the 8086, rather than the required four. This is a good example of the intelligence built into the models.

When we discovered this error, we stopped the simulation, corrected the stimulus file, and ran the simulation again. The second time we ran the simulation it did not show the RESET error. As directed by the trace settings we put in the PCL file, the simulator produced a trace of each state as the 8086 wrote to and read from memory. The bottom few lines of Figure 11-28 show some examples of the type of information the trace gives you. Note that the first operations the simulator carries out are to write to and read back from RAM locations as specified in the PCL file. A careful study of the trace showed that values were written to memory and read back correctly. This indicates that the address decoders are working correctly and that the circuit connections are correct. After we fixed the RESET problem described before, we ran the simulator again and got no significant timing warnings, so we felt reasonably sure the system would work correctly when we designed and built a PC board for it.

A very important point here is that it took only about 10 to 12 hours to design the system in Figure 11-25, draw the schematic for the system, and completely simulate it. Perhaps you can see that when designing a more complex system such as the microcomputer system in Figure 11-15*a*, simulation is the only practical way to determine if all the timing requirements are met in the design.

Design for Test

Once a system has passed simulation, the next step is to design in some circuitry which allows the system to be easily tested when it goes to production. Many



```
# Executing object named: '/idea/sys/lib/lsim_server.mod'
#   LOGIC SIMULATION SERVER V6.1_1.10 Monday, April 18, 1988 6:01:59 pm (PDT)
#   LAI Version: MG_A3_610_970_200  May 17, 1988
#   SmartModels: All pictorial, graphic, and audiovisual works, collective works
#                representations, compilations, and arrangements therof,
#                Copyright 1984-1988 Logic Automation Incorporated.
#
#   Note: Loading the PCL program from file "/user/doug/micro2/MICRO_OBJ".
#         Instance I$4(U2:I8086-2), sheet1 of micro2 at time 0.0
#
# ! Warning: Input pin MNMX is not allowed to change (will continue in MIN mode)
# !           Instance I$4(U2:I8086-2), sheet 1 of micro2 at time 0.0
#
#   Note: Loading the JEDEC file "/user/doug/micro2/U11.JED"
#         Instance I$62(U11:GAL16V8-15), sheet1 of micro2 at time 0.0
#         --- 173 fuses have been blown.
#
#   Note: Loading the JEDEC file "/user/doug/micro2/U8.JED"
#         Instance I$11(U8:GAL16V8-15), sheet1 of micro2 at time 0.0
#         --- 169 fuses have been blown.
#
#   Note: Loading the memory image file "/user/doug/micro2/RAM0EVEN"
#         Instance I$61(U10:MCM6164P70), sheet1 of micro2 at time 0.0
#         --- 257 values have been initialized.
#
#   Note: Loading the memory image file "/user/doug/micro2/RAM0ODD"
#         Instance I$41(U9:MCM6164P70), sheet1 of micro2 at time 0.0
#         --- 257 values have been initialized.
#
#   Note: Loading the memory image file "/user/doug/micro2/RAMFEVEN"
#         Instance I$63(U13:I27256-25), sheet1 of micro2 at time 0.0
#         --- 257 values have been initialized.
#
#   Note: Loading the memory image file "/user/doug/micro2/RAMFODD"
#         Instance I$15(U12:I27256-25), sheet1 of micro2 at time 0.0
#         --- 257 values have been initialized.
    VIEw Sheet
QuickSim>
    DO MICRO.DO
    RUN 100000
#
# ! Warning: RESET did not last 4 Clock Cycles.
# !           Instance I$4(U2:I8086-2), sheet1 of micro2 at time 1562.5
#
#   Trace: Trace is turned on
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 4812.5
#
#   Trace: Trace level is now set to 1 (internal timing states shown)
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 4812.5
#
#   Trace: CPU state T1
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 4937.5
#
#   Trace: Write Memory (1-byte) location 00000 with 0000
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 4937.5
#
#   Trace: CPU state T2
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 5312.5
#
#   Trace: CPU state T3
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 5687.5
#
#   Trace: CPU state T4
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 6062.5
#
#   Trace: CPU state T1
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 6437.5
#
#   Trace: Read Memory (1-byte) location 00000
#          Instance I$4(U2:I8086-2), sheet1 of micro2 at time 6437.5
```

FIGURE 11-28   Screen messages during simulator invocation and run.

microcomputers now contain built-in self-test (BIST) circuitry so that the unit does a complete internal test each time the power is turned on. If the unit fails any test, it sends a message to the CRT.

After the test circuitry is added, the circuit is simulated again to make sure the added test circuitry has not adversely affected the operation of the circuit.

Printed-Circuit-Board Design

In the old says we used a light table and large plastic sheets to develop the layout for a PC board. To produce "pads" for IC pins, transistor leads, resistor leads, etc., we stuck opaque "donuts" on the sheets. To produce traces between pads we used opaque tape. The plastic sheets were photographed and the resulting films were used to produce the desired patterns of traces on copper plated circuit boards.

Now we use automatic place-and-route programs such as Board Station from Mentor Graphics, Allegro from Valid Logic Systems, or Tango PCB from ACCEL Technologies, Inc., to lay out PC boards. These programs work with the netlist file and determine the best placement of components and the most efficient route for traces between components. The programs allow user interaction so that specific paths can be optimized if needed. For example, in designing a PC board for a very high speed system, you might determine the actual signal delays from an initial layout attempt and then resimulate the system with these delays. If the resimulation shows a problem, you can manually alter the layout to solve the problem before going on.

The file produced by the PC board layout program is sent to a laser printer to directly produce film negatives for each layer of the board. The negative is used to photographically produce the desired pattern on a copper-plated PC board. A chemical solution then etches copper from all the areas of the board except those where component pads, traces, ground planes, and power planes are desired. For a multilayer board, several individual boards are produced and then epoxied together under pressure to form a single board.

The board is then drilled under computer control. Finally, the plated-through holes and other *vias* which connect traces on different layers are electrochemically added to the board.

After manufacture, the "bare" PC boards are tested with a computer-based tester to check for shorts and opens. On a prototype PC board, minor problems can often be solved by, for example, drilling out a plated-through hole which accidentally got shorted to a power plane. A jumper wire can be added to make a missed connection.

Case Design

Once the PC board, power supply, and display have been designed for an instrument, the mechanical engineer can design the case for the system. A program such as the Mentor Graphics Package Station can be used to do much of this design. This program allows the designer to draw a three-dimensional view of a case and the placement of the components in the case. The Package Station program also a'lows a designer to determine the temperature that will be present at each location in the prototype case for a specified ambient temperature and airflow. This feature allows the designer to determine if the airflow is great enough, the placement of the PC board(s) in the case is reasonable, and perhaps if devices which produce a large amount of heat are placed too close together on a PC board. Here is another example of "software breadboarding," which saves much work and materials because, if a problem is found, you can simply go back to the computer screen and try a new design instead of producing a new physical box and trying it.

Developing the System Software

In addition to designing the hardware of a microcomputer, you also have to develop the BIOS software which allows programs to interact with the hardware. As we said earlier, the logic automation hardware verification model for a processor such as the 8086 allows you to include statements in a PCL file to initialize the programmable peripheral device models, write data to them, and read data from them. This is a way to verify the address, operation, and timing of these devices. If a fully functional model is available for the microprocessor, you can write sections of actual code for the microprocessor and run the code as part of the simulation.

When a prototype PC board for the system becomes available, an emulator such as we described in Chapter 3 can be used to develop the more complex software procedures of the BIOS.

Production and Test

Once the prototype of a system is debugged and any necessary changes are made, the design is finalized and released to production. Many parts of the production, testing, and troubleshooting of the instrument are done with the aid of computer programs.

Programs are available to generate a parts list from the netlist for a design. Other available programs direct a robot to collect the needed parts from the warehouse for the production run. A computer program running on an automatic tester tests the bare PC boards for shorts and opens before parts are inserted. Another program controls the machine that automatically places the components on the printed circuit board. The machine which solders all the components on the board is most likely controlled by a microcomputer program. Still another computer program controls the machine which automatically tests the finished PC boards. The program for this automatic test system uses test vectors which were developed as part of the design process. If the product does not have a complete built-in self-test, the finished product is also tested with an automatic test system. The linking together of all the computer-based tools used in the production of a product is called computer integrated manufacturing, or CIM.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

Motherboard and system expansion slots

8086 minimum mode and maximum mode

DMA operation

DMA channel

DRAM
$\overline{RAS}$ and $\overline{CAS}$ strobes
Refresh: burst and distributed modes
82C08 DRAM controller IC
Error detecting and correcting
Hard and soft errors
Parity check
Hamming codes and syndrome word
Page mode read/write access
Static column read/write access
Cache memory system
Direct-mapped cache
Two-way set associative cache
Fully associative cache

8087 math coprocessor
Data types and terms
Word, short, and long integers
Packed decimals
Short-, long-, and temporary-reals
Fixed-point numbers
Floating-point numbers
Normalizing
Significand, mantissa, exponent, biased exponent
Single- and double-precision representation

Electronic Design Automation
Schematic capture
Simulation
Gate-level model
Behavioral model
Hardware model
Time step
Stimulus file
Design for test
PC board layout
Case design
Computer integrated manufacturing

REVIEW QUESTIONS AND PROBLEMS

1. Why are microcomputers such as the IBM PC designed with peripheral expansion slots instead of having functions such as a CRT controller designed into the motherboard?

2. Describe how the control bus signals are produced for an 8086 system operating in maximum mode.

3. Why is DMA data transfer faster than doing the same data transfer with program instructions?

4. Describe the series of actions that a DMA controller will perform after it receives a request from a peripheral device to transfer data from the peripheral device to memory.

5. Describe how the 20-bit memory address for a DMA transfer is produced by the circuit in Figure 11-5.

6. Describe the function and operation of devices U5 and U6 in Figure 11-5.

7. Sketch the sequence of signals that must occur to read a data word from a dynamic RAM such as the TMS44C256.

8. List the major tasks that must be done to support dynamic RAM in a microcomputer system.

9. How does a dynamic RAM controller in a system such as that in Figure 11-9 arbitrate the dispute that occurs when the CPU attempts to read from or write to a bank of dynamic RAMs while the controller is doing a refresh cycle?

10. *a.* What timing parameter limits the rate at which data words can be read from random rows (pages) in a DRAM?
b. Explain how page mode operation of a bank of DRAMs makes it possible for a microprocessor to read data words without wait states.
c. What is the main difference between page mode operation and static column mode operation of a bank of DRAMs?

11. *a.* Describe how an SRAM cache reduces the average number of wait states required by a microprocessor which uses DRAM for its main memory.
b. How does a cache controller keep track of which blocks from the main memory are present in the cache?
c. With a direct-mapped cache system, what does each entry in the cache tag RAM represent?
d. In a direct-mapped cache system, only one block with a particular number can be present in the cache at a time. How does a two-way set associative cache overcome this problem?

12. Describe how parity is used to check for RAM data errors in microcomputers such as the IBM PC. What is a major shortcoming of the parity method of error detection?

13. When using a Hamming code error detection/correction scheme for DRAMs, how many encoding bits must be added to detect and correct a single-bit error in a 64-bit data word?

14. How can you tell from the schematic that the 8088 in Figure 11-23 is configured in maximum mode?

15. Device U7 in Figure 11-23 has a signal named AEN connected to its $\overline{OE}$ input. If, in troubleshooting this system, you find that this signal is not getting asserted, on which schematic sheet would you first look to see how this signal is produced?

16. In what ways are a standard microprocessor and a coprocessor different from each other?

17. *a.* Convert the decimal number 2435.5625 to binary, normalized binary, long-real, and temporary-real format.
    b. Why are most floating-point numbers actually approximations?

18. *a.* Which 8087 stack register is ST after a reset?
    b. Which 8087 stack register will be ST after one data item is read into the 8087?
    c. Describe the operation that will be done by the 8087 FADD ST(2),ST(3) instruction.
    d. How does the operation of the instruction FADDP ST(2),ST(3) differ from the operation of the instruction in 18c?

19. Describe the operation performed by each of the following 8087 instructions.
    a. FLD TAX_RATE
    b. FMUL INFLATION_FACTOR
    c. FSQRT
    d. FLDPI
    e. FSTSW CHECK_ANSWER
    f. FPTAN

20. Why does the assembler insert 9BH, the code for the 8086 WAIT instruction, before the code for most of the 8087 instructions?

21. Using the example program in Figure 11-24 as a guide, write an 8087 program which computes the volume of a sphere. The formula is $V = 4/3\pi R^3$.

22. *a.* When a coprocessor and a standard processor are connected together in a system such as that in Figure 11-23, why are the S2–S0 status lines, the QS1–QS0 lines, the address, and the data lines of the two devices connected directly together?
    b. Where does the 8087 coprocessor in Figure 11-23 get its instructions from?
    c. How does the main processor distinguish its instructions from those for the 8087 as it fetches instructions from memory?
    d. Describe how the 8087 and 8088 work together to load a long-real data item from memory to the 8087 ST.
    e. How does the 8087 in Figure 11-23 signal the 8088 that it needs to use the buses?
    f. How can you prevent the 8088 in Figure 11-23 from going on with its next instruction before the 8087 has completed an instruction? What hardware connection in Figure 11-23 is part of this mechanism?

23. *a.* Describe how a schematic is drawn using a schematic capture program.
    b. What are the major advantages of the schematic capture approach over the traditional drafting approach?

24. *a.* What is meant by the term *software breadboard*?
    b. Describe the major advantages of simulation over hardware prototyping.
    c. What information does the simulation model for a device contain?
    d. Briefly describe the steps involved in simulating a microcomputer such as the one in Figure 11-25.
    e. What information does simulation give you about a circuit such as the one in Figure 11-25?

25. Briefly describe the sequence of steps in the electronic design automation method of designing, debugging, and producing an electronic product such as a microcomputer.

CHAPTER 12

C, a High-level Language for System Programming

In the last chapter we introduced you to the operation of the motherboard hardware of a typical microcomputer system. Before we discuss the operation of system peripherals such as CRTs, hard disks, and telecommunications links, we need to introduce you to the languages and tools which are now commonly used to write application and system-level programs.

Up to this point in the book we have used assembly language for all the programming examples because we were working very close to the hardware. As we said earlier in the book, assembly language is appropriate for initializing peripheral devices, writing programs which manipulate a lot of hardware, or writing programs which have to execute very fast. Writing large system-level programs in assembly language is slow and tedious, so we usually write major parts of these programs in a high-level language such as Pascal or C.

As you are probably aware, there are many different high-level languages. For the high-level language programming examples throughout the rest of this book, we use the C language. We chose C because it is very widely used in industry, it is a good stepping-stone to a modern programming language called C++, and it is very easy to learn if you are already familiar with 8086-type assembly language programming.

To develop a system-level program, the overall design is broken down into a group of modules. A decision is then made whether each module can be written in a high-level language or must be written in assembly language. The high-level modules are written, debugged, and compiled to produce .OBJ files. Likewise, the assembly language modules are written, debugged, and assembled to .OBJ files. All the .OBJ files are then linked together to produce a .EXE file which can be run. This is basically the same process we described in Chapter 5 for writing multimodule assembly language programs. In this chapter we will first show you how to write some simple programs in C, and then we will show you how to write programs which contain both high-level language modules and assembly language modules.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Describe how the tools in an integrated programming environment such as Borland's Turbo C++ IDE are used to edit, compile, link, run, and debug C programs.
2. Describe the data types that are available in C.
3. Declare and initialize simple variables, arrays, and structures in C.
4. Implement standard programming structures such as IF-THEN-ELSE, SWITCH (CASE), WHILE-DO, DO-WHILE, and FOR-DO in C.
5. Declare, define, and call functions (procedures) in C programs.
6. Write C programs which implement simple algorithms.
7. Write simple programs which consist of C modules and assembly language modules.

INTRODUCTION—A SIMPLE C PROGRAM EXAMPLE

As we said before, it is very easy to learn the C programming language if you are already familiar with 8086 assembly language programming. To give you some feeling for how easy it is to make this transition and to give you an introduction to the general structure of a C program, we will first show how the cost-price array example program in Figure 4-23 can be written in C.

If you look back at the program in Figure 4-23, you will see that this program adds a profit of 15 to each of eight costs. More specifically, the program reads in a value from an array called COST, adds a profit of 15 to the value read in, and puts the computed price in the corresponding element of an array called PRICES.

Figure 12-1*a* on page 390 shows a simple C program which will perform basically the same operations and also write the results out on your computer screen. The first point to observe in this program is that any text enclosed between /* and */ is a comment, not part of the actual program. The next parts to look at in this program are the statements which define the data the program is going to work with. The statement int cost[ ]= {20,28,15,26,19,27,16,29,39,42}; declares an array of 10 integers called cost and initializes the 10 elements of the array with the specified values. This corresponds

```
/* C PROGRAM F12-01A.C */
/* COMPUTE THE SELLING PRICE OF 10 ITEMS */

#include <stdio.h>
#define PROFIT 15
#define MAX_PRICES 10

int cost[] = { 20,28,15,26,19,27,16,29,39,42 }; /* array of 10 costs */
int prices[10];                                 /* array to hold 10 prices */
int index;                                      /* variable to use as index */

main()
{
    for (index=0; index <MAX_PRICES; index++)   /* for loop to compute */
    prices[index] = cost[index] + PROFIT;       /* 10 prices */

    for (index=0; index <10; index++)           /* for loop to display results */
    priatf("cost = %d, price = %d, \n", cost[index], prices[index]);

}
```

(*a*)

```
cost = 20, price = 35,
cost = 28, price = 43,
cost = 15, price = 30,
cost = 26, price = 41,
cost = 19, price = 34,
cost = 27, price = 42,
cost = 16, price = 31,
cost = 29, price = 44,
cost = 39, price = 54,
cost = 42, price = 57,
```

(*b*)

FIGURE 12-1 (*a*) Simple C program to add profit of 15 to each of 10 items. (*b*) Printout of program results.

to the COST DB 20, . . . statement in the program of Figure 4-23. The statement int prices[10]; declares an array of 10 integers called prices. Since no values are given, the elements of this array are not initialized. Note that the C program statements are terminated with semicolons.

Program lines which begin with a # are *preprocessor directives*. These lines do not generate any code; they give instructions to the compiler. The #define PROFIT 15 line in Figure 12-1*a*, for example, tells the compiler to replace the name PROFIT with the constant 15 each time it finds PROFIT in the program. This is equivalent to the PROFIT EQU 15 line in the assembly language version in Figure 4-23. The #define MAX_PRICES 10 line in Figure 12-1*a* is another example. As we pointed out in our earlier discussions of assembly language programming techniques, it is very important to define constants at the start of a program in this way, rather than using "hard" numbers directly in the program. The reason is that if you have to change the number, you only have to change the value in the equ or the #define, instead of finding and changing the value each place it occurs in the program. Note that we always use uppercase letters for constants such as PROFIT, so that we can tell them from variables, which we put in lowercase letters.

The int index; statement in Figure 12-1*a* declares a variable called index. The int at the start of the statement indicates that the variable can have only integer values. This index will be used to point to the array element being processed at a particular time and to keep track of how many elements have been processed. Now that you have an overview of the data, let's take a look at the action part of the program.

All the action statements in C programs, even those in the mainline part of a program, are written in functions. In Pascal and some other languages, a function is the name given to a procedure which returns some value(s) to the calling program. In C all procedures are referred to as functions whether they return a value or not.

Every C program must have a function, usually called main, which gets called when your program starts executing. Other functions are called from main as needed. As you can see, the main( ) function in Figure 12-1*a* contains a for structure and two statements. You use the "curly braces" { and } to enclose the parts of a function. The parentheses ( ) after the name of the function are used to contain parameters and the names of variables that you want passed to the function. Later we will show you examples of how to do this. Empty parentheses after a function name mean that no parameters are being passed to the function.

The statement for (index=0; index <MAX_PRICES; index++) implements a FOR-DO loop which executes the statements contained in the second set of curly braces 10 times (index values of 0 through 9). The index++ term in the parentheses means that the value of index will be incremented each time through the loop. The prices[index]=cost[index]+PROFIT statement and the printf statement will also be implemented each time through the loop.

As perhaps you can figure out, the prices[index]=cost [index]+PROFIT; statement reads an indexed location in the cost array, adds a PROFIT of 15 to the value read, and writes the result to the same indexed location in the prices array. Note how the variable index is used here to access the elements in each array and also to determine how many times the loop executes.

Each time through the loop the printf statement calls the predefined printf( ) function which sends the specified text and values to the screen. The parentheses after printf contain the parameters we are passing to the function. Characters enclosed in " " inside the parentheses are printed out as written until a % is encountered. A % indicates that the value of a variable is to be inserted at that point. The name of that variable is included in a list of variables after the second " in the print statement. In this example the first variable encountered after the second " is cost[index], so the value of this variable will be printed out after cost= is printed out. The d after the % tells the function to print the decimal value of cost[index]. When the function encounters the second %d in the parentheses, it will print the decimal value of prices[index], the next variable after cost[index] in the variables list. The \n in the statement stands for "newline" and tells the printf function to send a carriage-return character and a linefeed character. This will move the cursor to the start of the next line down on the screen. Figure 12-1*b* shows the printout produced by the printf function when this program was run.

As you can see in Figure 12-1*a*, the printf function is not present in our program. The printf function is found in a library of input/output functions that comes with the program development software. The #include <stdio.h> at the start of the program tells the preprocessor part of the compiler that the prototype for the printf function is in a file called stdio.h. The linker will use this prototype to get the object code for the printf function from a library file and link it with the object code for our price.c program so that it will be part of the final executable file. In a later section we will tell you more about predefined functions.

If you compare the number of statements in our C program with the number of statements needed to do the job in the assembly language version in Figure 4-23, you should immediately see one of the advantages of writing as many programs as possible in a high-level language. In the next section we discuss some software tools you can use to develop your own C programs. Then in the following sections we show you much more of the structure and syntax of the C language.

PROGRAM DEVELOPMENT TOOLS FOR C

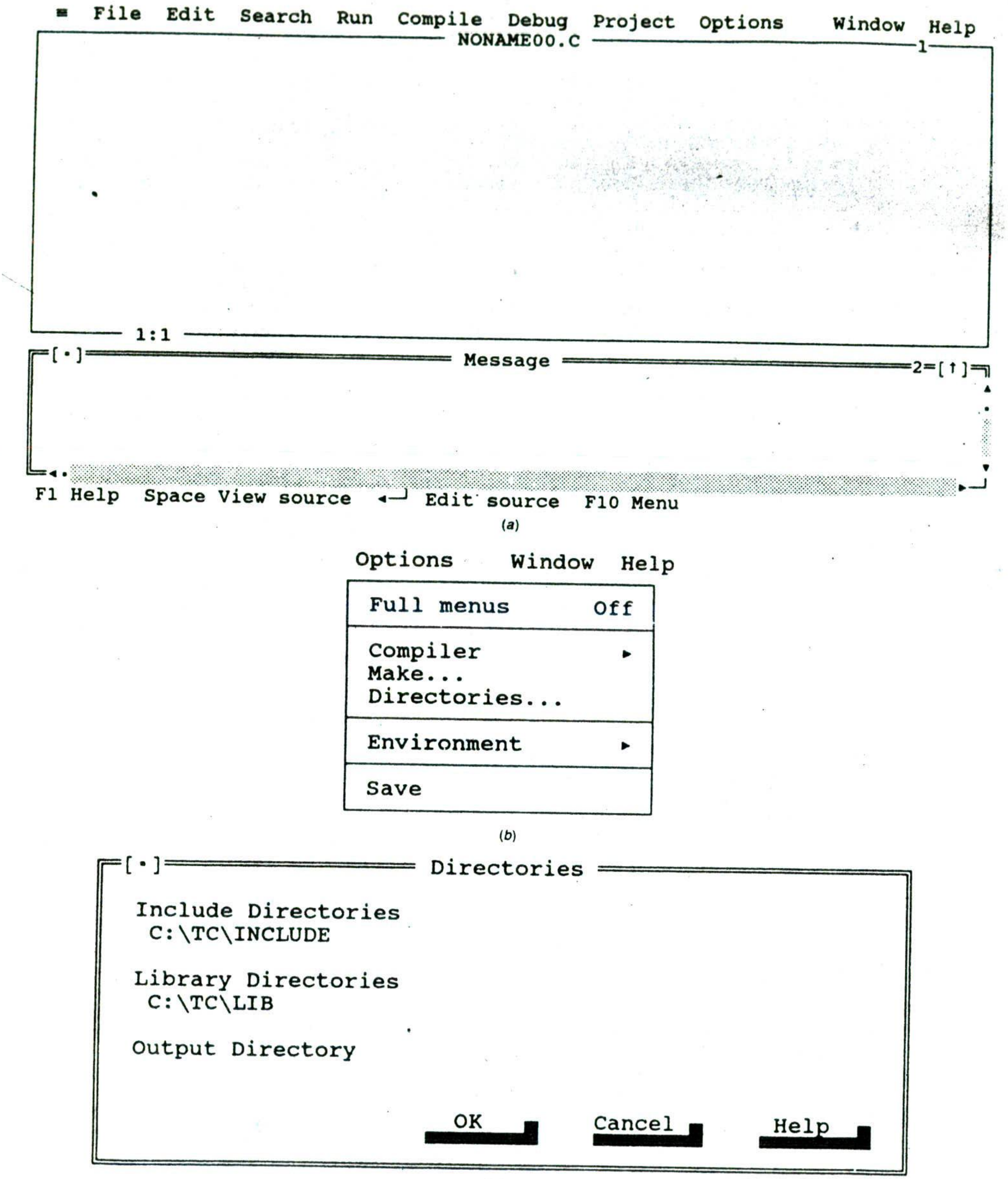
To develop a C program you need an editor to create a source program such as the one in Figure 12-1*a*, a compiler to convert the source program to an object code file, a linker to link the various object code modules of your program into an executable (.exe) file, and a powerful debugger to help you get the program working correctly. For the examples in this section we chose the Borland Turbo C++ Integrated Development Environment which has all these features and many more. As the name implies, the Turbo C++ tools also fully support AT&T C++ version 2.0 as well as ANSı standard C. We don't have the space or the need in this book to teach C++, but the Borland documentation contains a tutorial and many examples which will help you learn it if you want to later.

The term Integrated Development Environment means that you can access all the programming tools from one on-screen menu. We chose the Borland IDE system because it is very powerful but easy to use, the company has a generous educational pricing policy, and the company gives very good support if you encounter problems. Other available tool sets such as Programmer's Workbench from Microsoft are very similar, so you should have little trouble adapting the following discussion if you have some other set of programming tools.

The purpose of this section is not to make you an expert with the Borland tools but to show you enough about using tools such as these that you can enter, run, and experiment with the simple program examples in the later sections of the chapter. Even if you don't have tools such as these available, this section should show you how programs are developed in a modern programming environment and some of the features you should look for when you buy a toolset.

For the following discussions we assume your Borland Turbo C++ Integrated Environment tools and libraries are all installed in a hard-disk directory named C:\tc, as described in the Borland manual, and the path command in your autoexec.bat file contains C:\tc and c:\tc\bin. We further assume that your work disk is a floppy in the A drive. Here's how you use these tools to develop a program such as the SELL.C program in Figure 12-1*a*.

To bring up the turbo C environment you simply type tc and press the Enter key. After a short pause the main menu screen shown in Figure 12-2*a*, page 392, will appear. Each of the entries in the banner at the top of the display represents a pull-down menu of commands. To get to one of these menus you hold down the Alt key and press the letter key which corresponds to the first letter of the desired menu's name. If you have a mouse on your system, you can use the mouse to move the cursor to the desired menu box and click the left mouse key. Incidentally, almost all commands in the Turbo C environment can be executed by pressing ı "hot key"


≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME00.C
1
1:1
[•] Message 2=[↑]
F1 Help Space View source ◄┘ Edit source F10 Menu

(a)

(b)

Include Directories
C:\TC\INCLUDE

Library Directories
C:\TC\LIB

Output Directory

(c)

FIGURE 12-2 Borland Turbo C++ Integrated Development Environment screen displays. (*a*) Main menu and edit window 2. (*b*) Options submenu. (c) Directories submenu of Options submenu.

combination such as Alt-F1, which executes the command directly, or by working your way through a sequence of menus with a mouse or the arrow keys. When you are first learning a new system, the menu method helps you better learn the available features, so we will emphasize that approach. Here's an example of how you work your way through one of these menus.

One thing you have to do when you create a program is to tell the compiler, linker, etc., where to put the .obj and .exe files they create. You use a command in the Options menu to do this. To get to the Options menu you hold down the Alt key and press the O key. Figure 12-2*b* shows the Options menu that appears. The command you want in the Options menu is the Directories command. To get to the submenu for this command, you simply press the D key, and the window display shown in Figure 12-2*c* will appear. You want to assign an Output directory, so you hold the Alt key down and press the O key to get to that line.

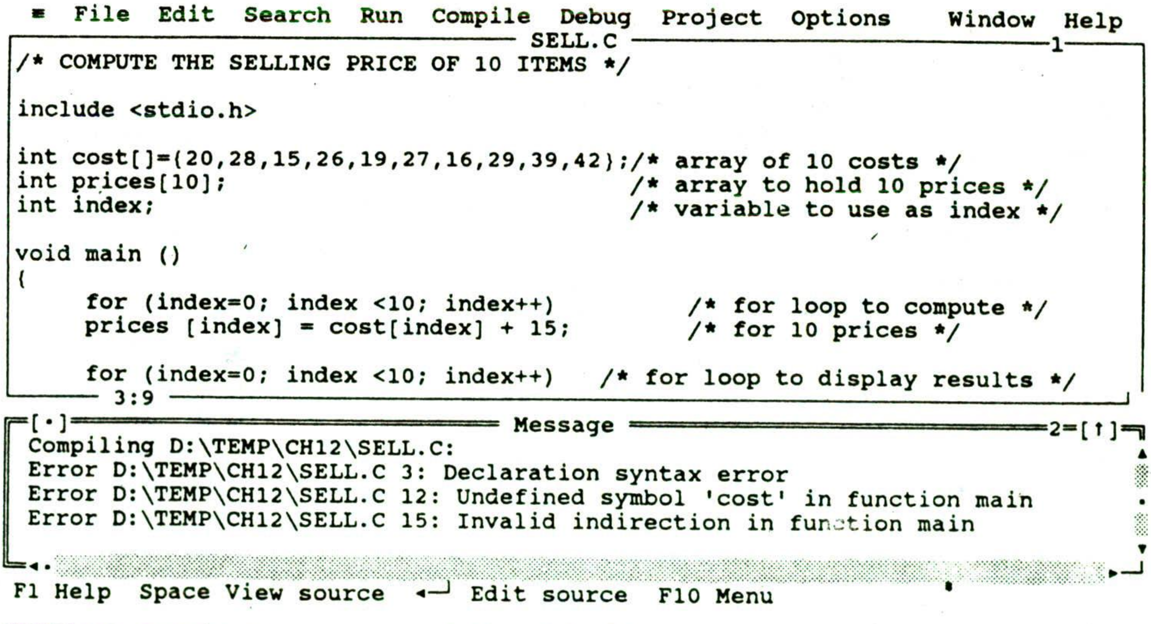
Now, suppose that you don't understand just exactly what you are supposed to put in this line. If you hit the F1 key, the system will display an explanation of the command requirements and, in some cases, examples. In this case the help window tells you that the .obj, .exe, and .map files will be stored in the location you enter as the output directory. After you have read the help window, you press the Esc key to close it.

Since you want your output files to go to your work disk in the A drive, you just type A: and press the Enter key. We don't expect you to remember all this; we just used it as an example of how you work your way down through a sequence of menus to perform a desired operation and how to get help with a command if you need it.

The next step in developing a program is to use the editor to enter the source text for the program. For a new program you go to the File menu and press the N key. A blinking cursor will appear in the large window on the screen and you can type in the text of your source program. To work on an old file you go to the File menu and press the O key. When a list of files appears, you select the desired file from the list and the file will be loaded into the editor window. In either case you edit the program just as you would with any text editor. You may find it helpful to use spaces instead of tabs to format your programs, because the default tab setting of most printers is 8, and with this setting C programs do not usually fit easily on 8.5-in.-wide paper. Incidentally, the IDE editor accepts Wordstar commands such as Ctrl g to delete a character, Ctrl t to delete a word, and Ctrl y to delete a line, so if you are familiar with Wordstar you should find the editor very easy.

After you type in the source file you need to save it on your work disk. To do this go to the File menu, select the Save As line and press the Enter key. When a small window appears, you can type in a file name such as a:SELL.C and press the Enter key.

The next step is to compile the program to generate the .obj file. To do this go to the Compile menu and press the C key. If the compiler finds any errors, it will display a window with a flashing error message. When you press a key, the error message(s) will be displayed in the message window at the bottom of the screen. Figure 12-3 shows the error messages we produced


≡ File Edit Search Run Compile Debug Project Options Window Help
SELL.C
1
3:9
[•]
Message
2=[↑]
F1 Help Space View source ◂┘ Edit source F10 Menu

```
/* COMPUTE THE SELLING PRICE OF 10 ITEMS */

include <stdio.h>

int cost[]={20,28,15,26,19,27,16,29,39,42};/* array of 10 costs */
int prices[10];                            /* array to hold 10 prices */
int index;                                 /* variable to use as index */

void main ()
{
     for (index=0; index <10; index++)          /* for loop to compute */
     prices [index] = cost[index] + 15;         /* for 10 prices */

     for (index=0; index <10; index++)    /* for loop to display results */
```

```
Compiling D:\TEMP\CH12\SELL.C:
Error D:\TEMP\CH12\SELL.C 3: Declaration syntax error
Error D:\TEMP\CH12\SELL.C 12: Undefined symbol 'cost' in function main
Error D:\TEMP\CH12\SELL.C 15: Invalid indirection in function main
```

FIGURE 12-3 Compiler error messages generated by omitting # in #include<stdio.h> directive.

when we intentionally left out the # sign in front of the include directive in SELL.C. A highlighted line in the source program indicates the statement which caused the error highlighted in the message window. You can press the Enter key to switch from the message window to the edit window and correct the error. For this error all we had to do was insert the missing # before the include directive. A major error such as this will cause many errors throughout the program, so when you find one of these it is a good idea to compile the program again before you start chasing down the other indicated errors.

To recompile the program all you have to do is go to the Compile menu and press the C key. Once the compile is successful, you should always save your source file before continuing. To do this just go to the File menu and press the S key. This is important, because if your program locks up the machine when you run it, your program will be lost.

The next step in developing a program is to generate an executable file which you can run. For this example the file will be given the name sell.exe by the linker which generates it. There are two ways to generate the .exe file. One way is to go to the Compile menu and press the M key to make a .EXE file. The second way to generate the .exe file is to go to the Run menu and press the R key to run the program. This sequence of commands generates the .exe file and also runs it. For a single-module program, this second method is obviously the easiest.

When the program has finished running, the display will return to your source program. If you wrote a program such as sell.c which outputs to the screen, this display will be left in an alternate screen buffer. To see the output of your program hold down the Alt key and press the F5 key. To toggle back to the IDE screen, hold down the Alt key and press the F5 key again.

Now, suppose that your program doesn't work correctly the first time you run it. The Turbo C++ IDE contains a powerful "source-level" debugger. A source-level debugger allows you to view your source program on the screen and single-step through it one statement at a time or run to a breakpoint you placed on a statement and watch the values of variables change as program statements execute. The debugger is integrated with the editor, compiler, and linker, so when you find an error, you can just go back to the edit window, fix the error, compile the program, and run the program again, all from the same main menu. In most cases this integrated approach is much more efficient than the independent tools approach. Here's a short example of how you might watch the values of some variables change as you single step your way through our example program, sell.c.

> NOTE: For this process to work as described, the program must have been just compiled and linked so the debugger has the needed "hooks."

As a first step let's assume that you want to observe the values of index, cost[index], and prices[index] change as you single-step through the program. To get ready to single-step you go to the Run menu and press the T key to Trace into the program. A highlighted bar will then appear on the first line of your program. To put a "watch" on each of the desired variables, you go to the Debug menu and press the W key to bring up the Watch submenu. Then press the A key to add a watch. When a small window appears, type in the name of the first variable you want to watch and press the Enter key. The name of the variable that you placed a watch on should appear in the Watch window at the bottom of the screen. You can repeat the procedure to put watches on other variables.

To execute the first line of the program, you press the F7 key. The highlighted bar will move to the next statement, and the values of the variables in the Watch window will be updated to show the results of executing that instruction. Figure 12-4 shows the result after stepping through the for loop in sell.c a couple of times.

If you want to determine the value of some variable that you did not put in the Watch window, you go to the Debug menu and then press the I key to execute the Inspect command. When a small window appears, you type in the name of the variable that you want to look at and press the Enter key. The current value of the specified variable will be returned in the Inspect window. You press the Esc key to get back to stepping through the program.

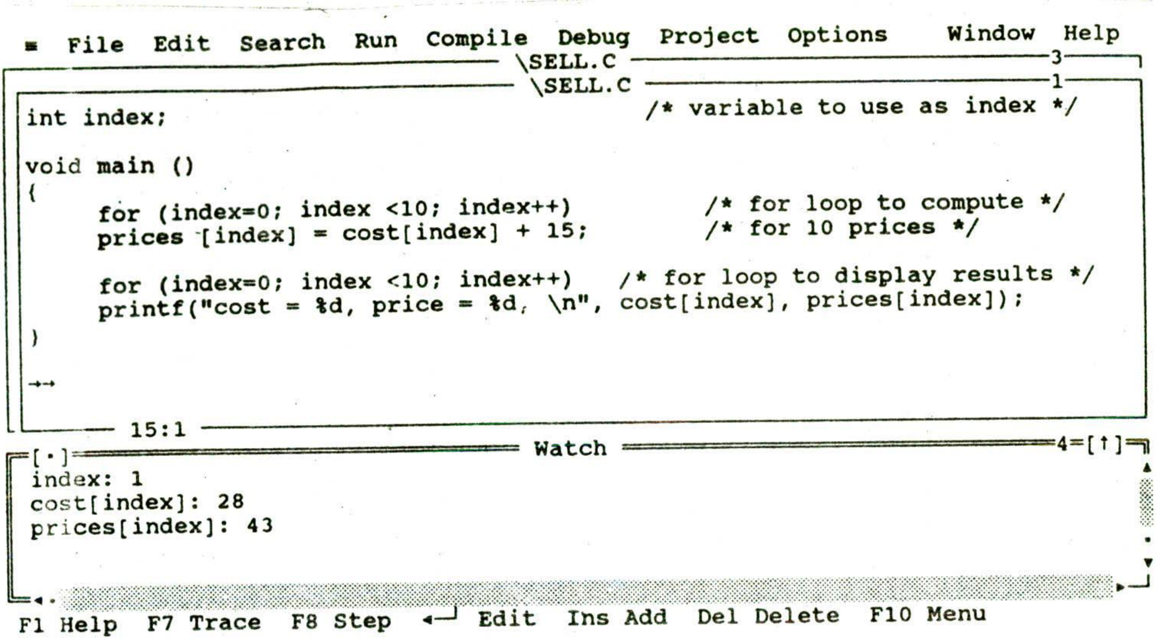
If you find an error as you step through your program, you can just edit the source code version. After you save the new version you can run the program again by simply going to the Run menu and pressing the T key. In this case the IDE tools will detect that the source program has been modified, and they will automatically compile, link, and put the highlighted bar on the first line of your program. You can then single-step through the program by just pressing the F7 key.

Again, the point here is not for you to remember all this key pressing, but to see how easy it is to use a source-level debugger and an integrated environment such as IDE to write and debug your C programs.

Before we leave this section, there is one additional point we want to mention. Modern compilers such as the one in the Turbo C++ IDE allow you specify how you want the generated object code to be optimized. In its default mode the compiler compiles your program to a binary instruction sequence which uses minimum memory. An alternative is to tell the compiler to produce code which is optimized for speed. You can also tell the compiler to make maximum use of registers to hold variables and to rearrange the code so that loops and other jumps are optimized.

We usually leave the compiler optimization in its default mode when debugging a program, and then when we know the program works correctly, we recompile it with speed, register, and jump optimizations "on" to produce the final version of the program. The reason we initially leave these optimizations off is that it is very difficult to step through a program which has been highly optimized unless you are familiar with the algorithms used by the compiler.

Now that you have an overview of the tools used to develop C programs, we will show you more of the


≡ File Edit Search Run Compile Debug Project Options Window Help
\SELL.C 3
\SELL.C 1
15:1
[•] Watch 4=[↑]
F1 Help F7 Trace F8 Step ↵ Edit Ins Add Del Delete F10 Menu

```
int index;                                    /* variable to use as index */

void main ()
{
     for (index=0; index <10; index++)            /* for loop to compute */
     prices [index] = cost[index] + 15;           /* for 10 prices */

     for (index=0; index <10; index++)    /* for loop to display results */
     printf("cost = %d, price = %d, \n", cost[index], prices[index]);
}
```

```
index: 1
cost[index]: 28
prices[index]: 43
```

FIGURE 12-4 Debugger screen display showing watch values.

structure and syntax of the C language so you can write some programs of your own.

PROGRAMMING IN C

Introduction

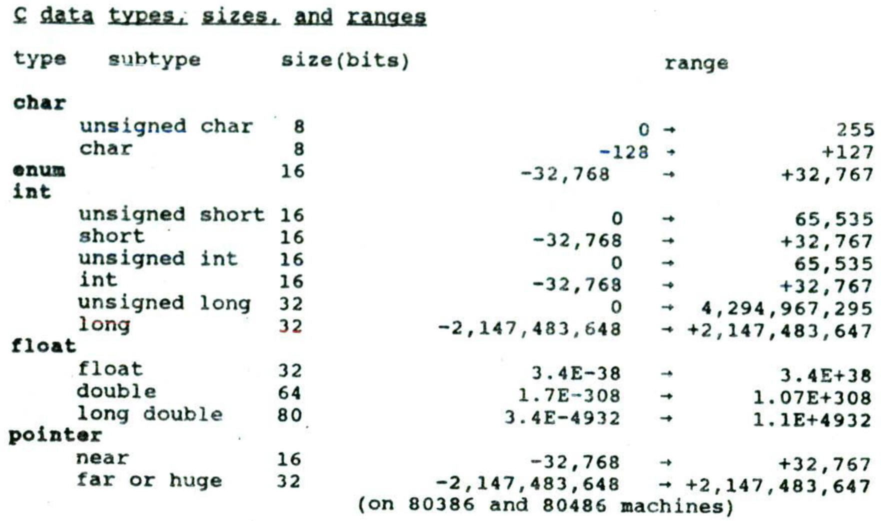
One reason it is easy to learn a second programming language is that you already know what features to look for. When you have to learn a new language, we suggest a "bottom-up" approach, roughly as follows.

1. First explore the data types that are available in the language and how these data types are represented. In 8086 assembly language, for example, you have worked with bytes, words, double words, and ASCII characters.
2. Then look at how basic statements such as variable declarations are written in the language. The DB, DW, DD, and array declaration statements are examples of this in 8086 assembly language.
3. Next, find out what logical, mathematical, and bit "operators" are available in the language. This is equivalent to looking at available 8086 instructions such as AND, ADD, INC, RCR, etc. It is best to just skim through these and pick out some commonly used ones. Don't try to remember them all the first time through.
4. Since you should always try to write programs in a structured way, the next step is to see how standard programming structures such as IF-THEN-ELSE, CASE, REPEAT-UNTIL, WHILE-DO, and FOR-NEXT are implemented in the language. Look for examples of these such as the 8086 assembly language examples we showed you in Chapter 4.
5. Most programs contain many procedures, so next find out how procedures are defined and called in the language and how parameters are passed to procedures. Look for some examples such as the assembly language examples we showed you in Chapter 5.
6. The final step in the discovery process is to use the new language to write some simple programs that you have written successfully in another language. Since the algorithms are already very familiar, all you have to do is determine the syntax needed to express the algorithms in the new language. You are really just translating each program from one language to another. The simple program in Figure 12-1*a* is an example of translating a familiar algorithm to C.

In the following sections we will lead you through the C language along the path described in the preceding steps. If you have some C programming tools available, we suggest that you work your way through the exercises at the end of the chapter and those in the accompanying lab manual to develop some skill in C.

C Data Types

In the first 10 chapters of this book you worked with integer data types such as bytes, words, double words, and ASCII character codes. Then in Chapter 11 you met a variety of floating-point data types. Figure 12-5 shows



C data types, sizes, and ranges

type	subtype	size(bits)	range
char			
	unsigned char	8	0 → 255
	char	8	-128 → +127
enum		16	-32,768 → +32,767
int			
	unsigned short	16	0 → 65,535
	short	16	-32,768 → +32,767
	unsigned int	16	0 → 65,535
	int	16	-32,768 → +32,767
	unsigned long	32	0 → 4,294,967,295
	long	32	-2,147,483,648 → +2,147,483,647
float			
	float	32	3.4E-38 → 3.4E+38
	double	64	1.7E-308 → 1.07E+308
	long double	80	3.4E-4932 → 1.1E+4932
pointer			
	near	16	-32,768 → +32,767
	far or huge	32	-2,147,483,648 → +2,147,483,647 (on 80386 and 80486 machines)

FIGURE 12-5 C data types and sizes.

the data types available in Turbo C, the number of memory bits used to store each type, and the range of values that can be represented by each type. You have met most of these data types before, so they should be readily understandable.

You use type char mostly for ASCII character codes. You use one of the six integer types to represent whole numbers according to the range needed. Likewise, you use one of the three floating-point types to represent real numbers, depending on the range of values you need for that variable. The C floating-point types shown in Figure 12-5 correspond to the three 8087 floating-point formats shown in Figure 11-18 and described in detail in the corresponding section of Chapter 11. Later in this chapter we will show you how the 8087 floating-point Pythagoras program in Figure 11-22 can be written in C.

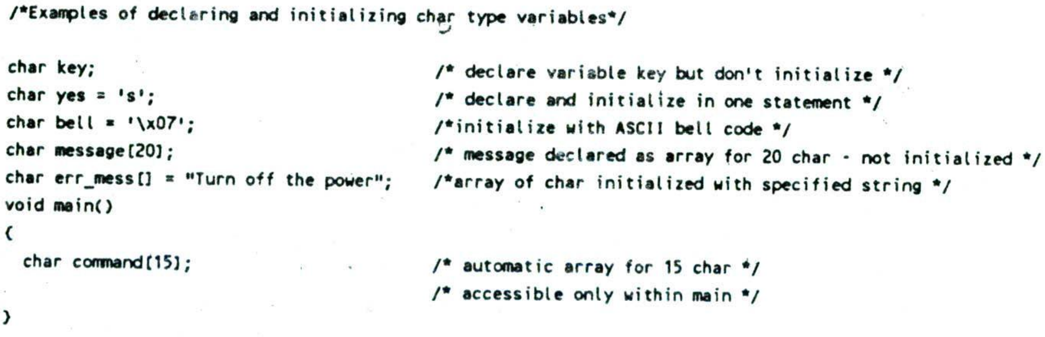
As in 8086 assembly language, C has near pointers and far pointers. You use a near (16-bit) pointer if you need to represent only the offset of a code or data word in a segment, and you use a far or huge (32-bit) pointer if you need to represent both the segment base and the offset of a code or data word. The *enumerated* data type shown as enum in Figure 12-5 is a user-defined type which can have integer values. You probably won't use this type in your first programs.

Declaring and Initializing Simple Variables in C

CHAR VARIABLES

In your 8086 assembly language programs you declared and initialized variables with DB, DW, and DD statements. The example program in Figure 12-1*a* showed you a few examples of how you declare and initialize simple variables and arrays in a C program. In this and the following sections we show how to declare and initialize variables of all the different C types. To start, Figure 12-6 shows some examples of how you declare and initialize char-type variables.

The first five variable declarations in Figure 12-6 are all *extern*, which means that they are outside of any function. Variables declared outside any function are "global," so they can be accessed by any function in a



```
/*Examples of declaring and initializing char type variables*/

char key;                               /* declare variable key but don't initialize */
char yes = 's';                         /* declare and initialize in one statement */
char bell = '\x07';                     /*initialize with ASCII bell code */
char message[20];                       /* message declared as array for 20 char - not initialized */
char err_mess[] = "Turn off the power"; /*array of char initialized with specified string */
void main()
{
  char command[15];                     /* automatic array for 15 char */
                                        /* accessible only within main */
}
```

FIGURE 12-6 Declaring and initializing char variables in C.

program. If you declare a variable within a function, the variable is by default *automatic*, which means that it is "local" and can be accessed only within that function. For example, the declaration "char command[15];" in Figure 12-6 is in function main, so the variable command is automatic and can be accessed only within main. Later, when we show you how to declare and use functions, we will discuss in more detail how you decide whether to make a variable extern or automatic. The general rule is to declare variables inside of main unless they *need* to be accessible to other program modules. This avoids a conflict if a module written by some other programmer has a different variable with the same name as one in the main module. Now, let's take a closer look at the syntax of declaring and initializing char-type variables.

As shown in Figure 12-5, a single char type variable uses 1 byte of memory. A declaration such as "char key" declares a variable named key and reserves 1 byte of storage for it. If the declaration is outside of main, the variable will be initialized with a default value of 0.

The second char example in Figure 12-6 shows how you can declare a variable named yes and initialize the variable with the ASCII code for a lowercase y. Note that the ASCII character is enclosed in single quotes.

If you want to initialize a char variable with the ASCII code for a nonprinting character, you can enter a \ followed by the hex code for the character. The \x07 in the third char example initializes the variable bell with the ASCII code which will sound a "bell" on your computer.

A char variable declaration such as one of the first three examples reserves space for just 1 byte, but the char message[20] example shows how you can declare an array of characters. In this case all 20 locations in the array will be initialized with the default value of 0.

The char message[ ]="Turn off the power"; statement in Figure 12-6 declares an array of characters and initializes the locations in the array with the ASCII codes for the characters enclosed in double quotes. Note that you use single quotes to initialize a single-character variable, but you use double quotes to initialize the elements in an array of characters. We did not need to put a number in the [ ], because the compiler automatically counts the number of characters enclosed in the double quotes and allocates the required memory bytes. The array actually contains one more byte than the number of characters in the string because the compiler automatically inserts an ASCII null character, 00H, as a sentinel after the last byte of the string. This sentinel character is used by many functions to identify the end of the string.

The declaration char command[15] in main in Figure 12-6 declares a 15-byte array of type char. As we said before, the declaration is in procedure main, so this array is automatic and can be accessed only in main. We did not initialize the array, so the locations in the command array will contain whatever random garbage happens to be in the memory locations set aside for the array. In some cases 0's get put in these locations, but you can't count on it, so you might tuck in a back corner of your mind that the default initialization for external arrays is 0 and the default initialization for automatic arrays is garbage.

INT VARIABLES

As shown in Figure 12-5, a simple INT variable uses 16 bits and can represent integers in the range −32,768 to +32,767. The example declarations in Figure 12-7 are all declared as simple int type, but you can replace the int at the start of any of these with one of the other int types shown in Figure 12-5 to get the range you need for a specific application.

The first int example in Figure 12-7 declares an int-type variable named headcount, reserves a 16-bit word in memory for it, and leaves the location initialized with a default value of 0. Headcount is declared outside of main, so it is extern.

The second int example shows you how to initialize a variable to 10 decimal, and the third int example shows you how to initialize a declared variable with FFFFH. The 0x in front of the ffff tells the compiler that the ffff represents a hexadecimal number. Note that since type

```
/* Examples of declaring and initializing int type variables */

int headcount;                            /* declare but don't initialize variable headcount */
                                          /* headcount is extern */
void main()
{
int index = 10;                           /* declare and initialize with 10 decimal */
int address = 0xffff;                     /* initialize with hex ffff */
int i=10, j=20, k=30;                     /* declare and init 3 int variables */
int prices[10];                           /* array of 10 int, uninitialized */
int cost[]={20,28,15,26,19,27,16,29,39,42};/* array of 10 int, initialized with values shown */
int test_scores[25][4];                   /* 2 dimensional array with 25 rows and 4 columns */
int av_temp[5][12][31];                   /* Three dimensional array- pages, rows, columns */
unsigned long population_1990;            /* 32-bit unsigned integer */
}
```

FIGURE 12-7 Declaring and initializing int variables in C.

int represents a signed value, ffffH is actually equal to −1. If you want to declare a variable and initialize it with a value of +ffffH, you can use a statement such as "unsigned int hex_value = 0xffff;."

The int i = 10, j = 20, k = 30; example in Figure 12-7 shows how you can declare and initialize three or more variables of the same type in a single statement to make your program more compact.

The following two examples declare arrays. These examples should be very familiar to you from the program in Figure 12-1*a*. The int prices[10]; statement declares an array of 10 words and leaves the 10 locations uninitialized. The int cost[ ] = {20,28,15,26,19,27,16,29,39,42}; declares an array of 10 words and initializes the 10 locations with the specified values. Note that you do not have to include the length of the array in the [ ] for the cost declaration, because the compiler counts the number of specified values and makes the array long enough to hold that number.

The last two int examples in Figure 12-7 show how to declare two- and three-dimensional arrays. A two-dimensional array consists of rows and columns. An instructor's grade roster is an example of a two-dimensional array. The rows represent the names of the students and the columns represent the scores on tests, quizzes, and labs. The statement int test_scores[25][4]; declares a two-dimensional array that might be used to store 4 test scores for each of 25 students. The 25 in this declaration represents the number of rows and the 4 represents the number of columns. We did not initialize this array, because a program that uses this array would probably prompt the instructor to enter the values for the array from the keyboard. To see how to initialize a two-dimensional array as part of the declaration, see the array declarations under the float type in Figure 12-8.

You can think of the three-dimensional array declared by the int av_temp[5][12][31]; statement in Figure 12-7 as consisting of 5 "pages" with 12 horizontal rows and 31 vertical columns on each page. This array represents the form in which the average temperature values for a 5-year period might be stored. The program that uses this array would probably compute the value for each element in the array using maximum and minimum values entered by a friendly weatherperson. Later we show you how to access elements in multidimensional arrays such as this.

FLOAT VARIABLES

As shown in Figure 12-5, the three floating-point number types available in C are float, double, and long double. The basic format of float-type declarations is the same as that for int type declarations, so we have just shown a couple of examples of this type in Figure 12-8. The first example again shows how you can declare and initialize several variables in a single statement. The second example shows how you can declare and initialize a two-dimensional array of real numbers. Note how the inner curly braces are used to set off the rows and the outer curly braces are used to enclose all the rows. The final float example in Figure 12-8 shows how to declare a long-double-type variable. I suppose that we will have to create a new floating-point type when the national debt becomes too large to be represented with a long-double-type variable.

Declaring, Initializing, and Using Pointers in C

INTRODUCTION

People who have not worked with an 8086-type assembly language often have trouble understanding pointers when they are first learning C. By now you have several chapters of experience with 8086 assembly language pointer instructions such as MOV AX,[BX] and MOV AL,COST[BX], so if we do our job well, you should have little trouble with C pointers.

To help you with the transition to C, we will not only show you how to declare and initialize pointers, we will show you how they are used in simple programs. To further help you, we will show the 8086 assembly language equivalents for some of the C examples we use. Read through this section until you thoroughly understand it, because much of the power of the C language is based on the use of pointers.

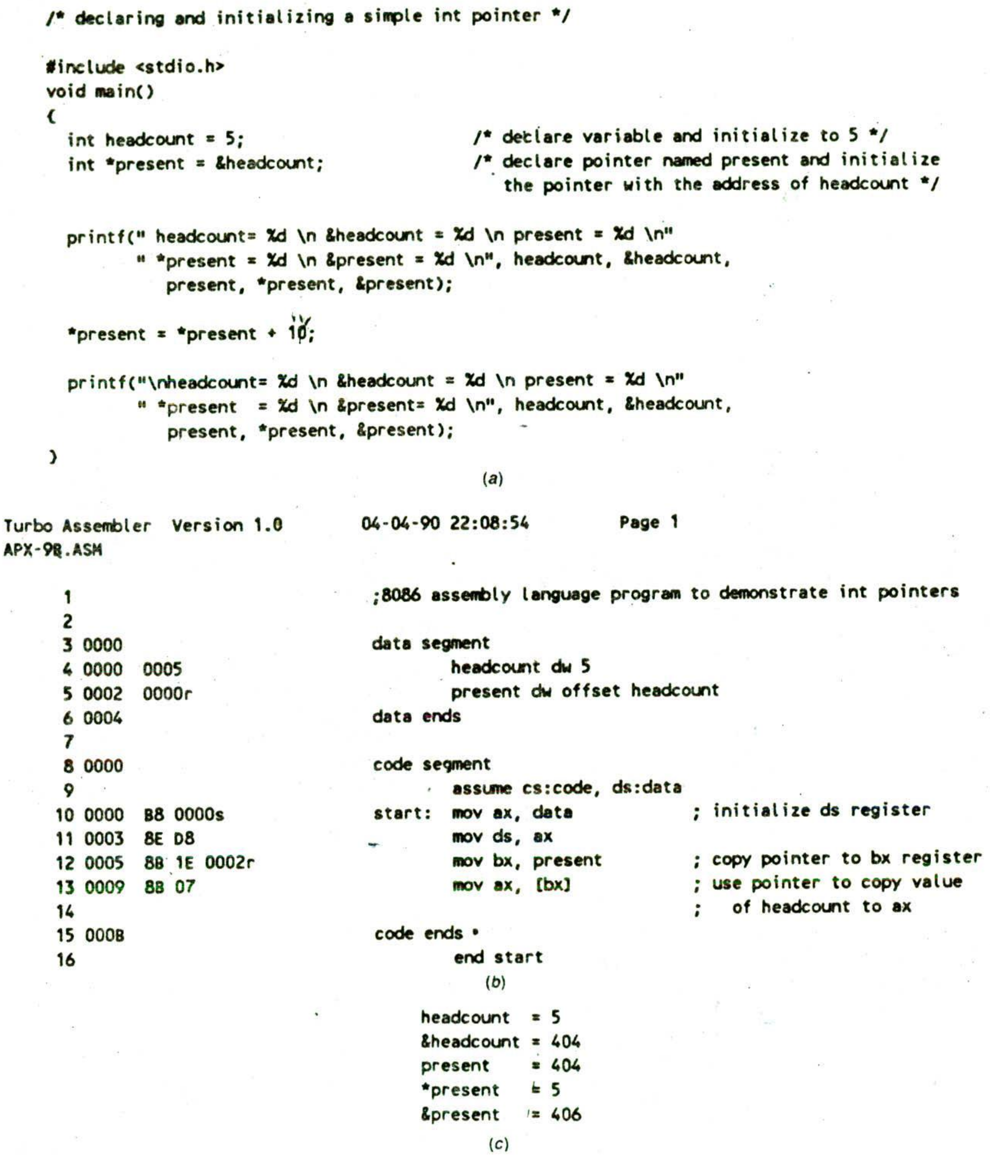
A SIMPLE INT POINTER

The first statement in the pointer example in Figure 12-9*a* declares an integer-type variable called headcount and initializes the variable with a value of 5. The second statement in this example declares a pointer named present and initializes the pointer with the address of

```
/* Examples of declaring and initializing float type variables */

long double national_debt;                    /* 80-bit floating point number */
                                              /* extern, anybody can access */
void main()
{
float side_a = 3.0, side_b = 4.0, side_c;     /* Init side_a and side_b, but not side_c*/
float max_min_temp[2][7] =
    {{37.3,42.0,42.9,46.0,51.7,44.2,40.0},
    {29.4,32.2,30.1,34.2,37.2,36.1,32.3}};    /* declare and initialize 2 dimensional
                                                 array of 2 rows and 7 columns */
}
```

FIGURE 12-8 Declaring and initializing float variables in C.



```
/* declaring and initializing a simple int pointer */

#include <stdio.h>
void main()
{
  int headcount = 5;                    /* declare variable and initialize to 5 */
  int *present = &headcount;            /* declare pointer named present and initialize
                                           the pointer with the address of headcount */

  printf(" headcount= %d \n &headcount = %d \n present = %d \n"
         " *present = %d \n &present = %d \n", headcount, &headcount,
           present, *present, &present);

  *present = *present + 10;

  printf("\nheadcount= %d \n &headcount = %d \n present = %d \n"
         " *present  = %d \n &present= %d \n", headcount, &headcount,
           present, *present, &present);
}
```

(*a*)

```
Turbo Assembler  Version 1.0        04-04-90 22:08:54         Page 1
APX-98.ASM

      1                             ;8086 assembly language program to demonstrate int pointers
      2
      3 0000                        data segment
      4 0000  0005                         headcount dw 5
      5 0002  0000r                        present dw offset headcount
      6 0004                        data ends
      7
      8 0000                        code segment
      9                                    assume cs:code, ds:data
     10 0000  B8 0000s              start: mov ax, data          ; initialize ds register
     11 0003  8E D8                        mov ds, ax
     12 0005  8B 1E 0002r                  mov bx, present       ; copy pointer to bx register
     13 0009  8B 07                        mov ax, [bx]          ; use pointer to copy value
     14                                                          ;    of headcount to ax
     15 000B                        code ends
     16                                    end start
```

(*b*)

```
headcount  = 5
&headcount = 404
present    = 404
*present   = 5
&present   = 406
```

(*c*)

FIGURE 12-9 (*a*) Declaring and initializing a simple int pointer. (*b*) Assembly language example of initializing and using int pointer. (*c*) Results produced by printf statement in Figure 12-9*a*.

the variable headcount. There are three points to store in your mind from this example.

First, the type for a pointer is the type of the data pointed to. Second, the * in front of the name present in the declaration tells the complier that present is a pointer. Third, the & in front of headcount is the "address of" operator. This operator tells the compiler that you want to initialize the pointer present with "the address of" the variable headcount. To summarize then, this statement declares a pointer-type variable named present and initializes it with the address of the int variable called headcount.

To help you relate all this to your previous experience, the DATA SEGMENT in Figure 12-9*b* shows the 8086 assembly language equivalent for these two C declarations. As you can see by the comments, the first DW statement sets aside a word of storage for (declares) HEADCOUNT and initializes headcount with 5. The second DW statement sets aside a word of storage for (declares) a variable named PRESENT and initializes it with the OFFSET of the variable HEADCOUNT. Since the offset of a variable is its address within a segment, PRESENT is then a pointer to HEADCOUNT. To access HEADCOUNT you can copy the pointer into BX with the instruction MOV BX,PRESENT and then copy the value of HEADCOUNT into AX with the instruction MOV AX,[BX].

The thought that probably comes to mind now is, Why didn't you just MOV AX,HEADCOUNT to get the value of HEADCOUNT into AX instead of using the indirect method? The answer is that for this case you can use the direct approach, but for other cases we show you later, you can't. To make sure you understand what is pointed to and what is doing the pointing in the declarations in Figure 12-9*a*, let's look at the actual values produced by the terms in the C declarations.

Figure 12-9*c* shows the results produced by the printf statement in Figure 12-9*a*. Work your way carefully through these so you see the two ways of representing the value of headcount and the two ways of expressing the address of headcount in programs.

You can refer to the value of headcount directly by name or with the *present representation. When used in a program statement, the * in front of a pointer name means "the contents of the memory location pointed to by that pointer." The *present term in the printf statement then means the value of the variable pointed to by present. The standard programming jargon for using a * in front of a pointer to refer to the value pointed to by the pointer is *dereferencing the pointer*. As you can see in our example here, present points to headcount, so the value printed for *present is the same as that printed for headcount. The only confusion here is that in a pointer declaration the * is used to indicate that the declared variable is a pointer, and in other program statements the * means the value of the variable pointed to by the pointer named after the *. The * in this context means the same thing as the [ ] in an assembly language statement such as MOV AX,[BX]. Remember how we told you to read these [ ] as "the contents of the memory location(s) pointed to by the value in the BX register."

The output of the printf function in Figure 12-9*c* also shows that you can represent the address where headcount is stored in two ways. One way is with the "address of" operator, &. The term &headcount is a shorthand way of saying "the address where the variable headcount is stored in memory." In its default mode the compiler assigns the same segment base address to your data segment, stack segment, and extra segment, so the address produced by printf for &headcount is just the offset of headcount in the segment. The second way to refer to the address of headcount is with the pointer, present. In the int *present = &headcount; statement, we declared present as a pointer and "pointed it" at headcount. As shown by the printout, the value of present then is the same as the value of &headcount.

Finally, in the printf output note the value produced by &present. This value represents the memory address where the compiler decided to store the pointer present. If you take another look at the assembly language equivalent of this program in Figure 12-9*b*, you can see how this corresponds to the offset where PRESENT is stored in the data segment.

Now that you know a little about C pointers, we want to take a moment to show you one reason why they are important.

When you call a C function you often want to pass parameters (arguments) to the function. The statement printf("%d", sum);, for example, calls the printf function and passes it a variable called sum. What is actually passed to the function is a *copy* of the variable sum. The function then is given the value of the variable sum but is not given access to the actual variable itself. The technical term for this method is *passing by value*. If only the value of a variable is passed to a function, then the function cannot modify the actual variable. For a function such as printf this is no problem, because printf is not intended to change the values of variables.

However, if you call a function which is intended to change the value of a variable, you must pass the address of the variable to the function. The function can then use the address it receives to access and change the value of the variable. As an example of this, the C statement scanf("%d", &headcount); calls the predefined function scanf to read a decimal value from the keyboard and assign the value to a variable called headcount. The &headcount in this statement passes the address of headcount to scanf so that scanf can write the new value in headcount. If present has been previously declared and initialized as a pointer to headcount with int *present = &headcount; statement, then another way to write the scanf statement is scanf("%d", present);.

In a later section we discuss C functions in great detail, but to give you a head start you might store in your mind the fact that when you call a C function to change the value of a variable, you must pass the address of the variable to the function instead of passing the value of the variable. The technical term for this is *passing by reference*. For a simple variable such as headcount, you can use &headcount to represent the address of the variable headcount. However, for many applications, declaring a separate pointer is a much more versatile technique. In the following section we show you how to declare and use pointers with simple, one-dimensional arrays. Throughout the rest of the book we will show many more examples of how you use pointers in programs.

USING POINTERS WITH INT ARRAYS

The program in Figure 12-10*a* shows three different ways to add a profit to each of 10 costs from an array of ints. As in the example in Figure 12-1*a*, we first declare an int array called cost, initialize the cost array with 10 values, and declare an empty array of 10 elements to hold the computed prices. In the third line of declarations we declare a simple variable called index and declare two pointers. The *cpntr = cost term declares a pointer called cpntr and initializes the pointer with the address of the first element in the array cost. The *ppntr = prices term in the third line declares a pointer called ppntr and initializes it with the address of the first element in the prices array. Now let's look at the three methods of accessing the elements in these arrays.

The first method shown in Figure 12-10*a* is the array-index method we showed you in Figure 12-1*a*. When you declare an array such as cost[ ], C treats the name cost as a pointer to the first element in the array.

```
/* C PROGRAM F12-10A.C - COMPUTE THE SELLING PRICE OF 10 ITEMS */
#include <stdio.h>
#define PROFIT 15
#define MAX_PRICES 10

void main()
{
 int cost[] = { 20,28,15,26,19,27,16,29,39,42 };
 int prices[10];
 int index, *cpntr = cost, *ppntr = prices;

 /* array index method */
     for (index=0; index <MAX_PRICES; index++)
     {
     prices[index] = cost[index] + PROFIT;
     printf("cost = %d, price = %d, \n", cost[index],
             prices[index]);
     }
 /* pointer method */
     for (index =0; index<MAX_PRICES; index++)
     {
     *ppntr = *cpntr + PROFIT;
     printf("cost = %d, price = %d,\n",*cpntr,*ppntr);
     cpntr++; ppntr++;
     }
 /* pointer arithmetic method */
     for (index =0; index<10; index++)
     {
     *(prices +index) = *(cost +index) + PROFIT;
     printf("cost = %d, price = %d, \n",
     *(cost + index), *(prices + index));
     }
```

(a)

```
;8086 PROGRAM F12-10B.ASM
;ABSTRACT : Assembly language program to add profit to costs using pointers
          ; Program adds a profit factor to each element in a
          ; COST array and puts the result in an PRICES array.

PROFIT   EQU      15H        ; profit = 15 cents
ARRAYS   SEGMENT
         COST     DB  20H,28H,15H,26H,19H,27H,16H,29H,39H,42H
         PRICES   DB  10 DUP(0)
         CPNTR    DW OFFSET COST
         PPNTR    DW OFFSET PRICES
ARRAYS   ENDS

CODE     SEGMENT
         ASSUME  CS:CODE, DS:ARRAYS
START:   MOV  AX, ARRAYS ; Initialize data segment
         MOV  DS, AX     ; register
         MOV  CX, 0010   ; Initialize counter

DO_NEXT: MOV  BX, CPNTR  ; Load cost pointer in BX
         MOV  SI, PPNTR  ; Load price pointer in SI
         MOV  AL, [BX]   ; Get element pointed to by CPNTR
         ADD  AL, PROFIT ; Add the profit to value read
         DAA             ; Decimal adjust result
         MOV  [SI], AL   ; Store result at location pointed
                         ;  to by PPNTR in PRICES
         INC  CPNTR      ; Increment pointers
         INC  PPNTR
         LOOP DO_NEXT    ; If not last element, do again
CODE     ENDS
         END  START
```

(b)

FIGURE 12-10 (a) C program which uses pointers to compute selling prices. (b) 8086 assembly language equivalent of program in 12-10a.

This pointer, however, is a constant, so it cannot be incremented to access the other elements in the array. To access the other elements in the array, you have to in some way add an index to cost. The term cost[index] in the array-index example in Figure 12-10*a* tells the compiler to generate the effective address by adding the value of index to the address represented by the name cost. Likewise the term prices[index] tells the compiler to generate the effective address of the variable by adding the value of index to the address represented by the name prices. The first time through the for loop, index will have a value of zero, so the first element in each array will be accessed when the prices[index]=cost[index] +profit; statement is executed. The second time through the for loop, index will have a value of 1, so the second element in each array will be accessed.

The method described for this C example is exactly the same method we used to access the array elements in the assembly language program in Figure 4-23. If you look back at that program, you will see that we loaded the index in BX, used the instruction MOV AL,COST[BX] to copy an element from the cost array to AL, and used the instruction MOV PRICES[BX],AL to copy the computed price back to the indexed location in the PRICES array.

The second method of accessing the elements in our arrays uses the pointers, cpntr and ppntr, that we declared. The statement *ppntr=*cpntr+profit; says read the value pointed to by cpntr, add a profit of 15 to the value, and write the result at the location pointed to by ppntr. In the initial declarations we initialized cpntr with the address of the first element in cost and ppntr with the address of the first element in prices. Therefore, the first execution of the for loop will read the first element in cost, perform the specified computation, and write the result in the first element of prices. The cpntr and ppntr pointers are variables, so they can be incremented, decremented, added to, subtracted from, etc., to access other elements in the arrays. The cpntr++; statement increments cpntr to point to the next element in cost, and the ppntr++; statement increments ppntr to point to the next element in prices.

NOTE: Both cpntr and ppntr were declared as pointers to int type variables, so the compiler automatically generates instructions which increment the pointers as needed to access the next elements in the two arrays. Since int variables take 2 bytes, the compiler will generate instructions which add 2 to the value of cpntr and add 2 to the value of ppntr. The next time through the for loop then, cpntr will point to the second element in cost and ppntr will point to the second element in prices. With this pointer method you do not need [index] to identify the desired elements in the arrays, because the pointers are incremented to point to the desired elements. The printf( ) statement in the pointer method example in Figure 12-10*a* also uses the *cpntr notation to represent "the contents of the memory location(s) pointed to by cpntr" and *ppntr to represent "the contents of the memory location pointed to by ppntr." We didn't bother to show you, but this second method produces the same printout as that shown in Figure 12-1*b*.

To help you further understand how this pointer version of the program works, Figure 12-10*b* shows how you could write it in 8086 assembly language. The program in Figure 12-10*b* actually generates machine code very close to that generated by the compiler for the C pointer example we have just discussed except that it works with bytes instead of words, and it obviously does not produce the code for the printf function.

In this assembly language example you can see that we first use a DW statement to declare and initialize a pointer to the first element in cost and another DW statement to declare and initialize a pointer to the first element in prices. Then in the code section of the program we load CPNTR into BX and PPNTR into SI so we can use them to access the arrays. We use MOV AL,[BX] to read in an element from cost and MOV [SI],AL to copy the result to prices. We then increment the pointers so they point to the next elements in the arrays and loop back to do the next add and store operation.

The third method of accessing the elements in the two arrays is the pointer arithmetic method shown in Figure 12-10*a*. As we said before, the name of an array such as cost is a pointer to the first element in the array. To access the other elements in the array you need to add an offset to the value of cost. One way to indicate this addition is with an expression such as cost[index] that we showed you in the first array-access method. Another way to indicate this addition is with an expression such as (cost+index). Putting a * in front of this expression gives *(cost+index), which translates to "the value in memory pointed to by the sum of cost+index." The expression *(cost+index) is exactly equivalent to the expression cost[index] and the compiler generates the same code for each expression. Note that since cost is a pointer to an int-type array, the compiler generates instructions which add two times the value of index to cost when it translates the expression (cost+index).

Now that you have seen the three methods of accessing an array, the question that may occur to you is, Which one is best? The answer is that usually you can use any of them. The array-index method is probably more intuitive when you are first learning about arrays, but most experienced C programmers use the direct pointer or the pointer arithmetic method because they generate considerably more efficient machine code. If for no other reason, you should use these last two methods in your programs so that you can easily follow them in other people's programs.

Another point we want to briefly make about the program in Figure 12-10*a* is the format in which the data is stored and manipulated. Cost is declared as type int, so according to Figure 12-5, 2 bytes are set aside for each element in cost. The compiler converts the decimal value supplied for each element to a 16-bit signed equivalent. When the program is loaded into memory to be run, these 16-bit signed values are loaded in the memory locations allocated for cost. When the program is run, the binary equivalent of 15 is added to

FORMAT SPECIFIER SYMBOL	PRINT
%d	decimal integer
%u	unsigned integer
%ld	long decimal integer
%p	pointer value
%f	floating point format
%6.2f	floating point format round off to two digits of decimal point, total of six digits
%e	exponential format floating point
%c	ASCII character for value
%s	string
%x or %X	hex value of integer

FIGURE 12-11 C format specifiers for use in printf, scanf, and other library functions.

each value from cost and the 16-bit signed result is put in the appropriate location in prices. The %d format specifiers in the printf( ) statement cause the printf function to convert the *cpntr and *ppntr values to their decimal equivalents before sending the values to the screen. The result is the decimal printout shown in Figure 12-1*b*. For reference Figure 12-11 shows the formats for some of the specifiers you can use with printf, scanf, and other predefined functions.

A FLOAT POINTER EXAMPLE

By now you are probably getting tired of the cost-price example, but we will use it one more time to quickly show you a few useful techniques that make the program more realistic.

Figure 12-12, page 404, shows the new "improved" version. The first improvement is to make the program able to work with floating-point numbers instead of just integers. We did this by declaring the two arrays as type float instead of type int.

The second improvement is to make it easy to change the program so it can work with some number of values other than 10. Note how we use the preprocessor directive #define MAX_PRICES 10 at the start of the program to declare a constant called MAX_PRICES and then use MAX_PRICES in the for loops and every time we refer to the number of elements in the arrays. If we want to change the number of elements in the arrays, all we have to do is change the value of MAX_PRICES in the #define and recompile the program. The compiler will automatically replace each occurrence of MAX_PRICES with the new value. This shows the advantage of using defined constants instead of hard numbers in a program.

A third improvement in the program is to add a section which allows you to enter any desired values instead of using just the fixed values we put in cost for the previous examples. To do this we declare the array cost as shown, but we do not initialize the array with fixed values. After using the predefined function printf to send a prompt message to the user, we use another predefined function called scanf and a for loop to read in 10 values entered on the keyboard.

The actual code for the scanf function is contained in a library file. The #include <stdio.h> preprocessor directive at the start of the program tells the compiler to look in the file stdio.h for the prototype of the scanf( ) function. When the program is linked, the code for scanf( ) and printf( ) functions will be linked with the code for the rest of the program to generate the executable (.exe) file.

The scanf("%f", cpntr) statement in the program calls the function and passes the parameters needed by the function. The scanf function needs to know what type of data you want it to read and where you want it to put the data. As with the printf function we used before, you use a format specifier to indicate the type of data you want it to read. In this program we want scanf to read floating-point values, so we pass a %f to scanf by putting it first in the ( ). As we said earlier, the scanf function requires that you pass it a pointer to tell it where to put the data read. We want the data values to be put in the cost array, so we pass the pointer cpntr that we initialized with the starting address of the cost array. Each time through the for loop, cpntr will be incremented so that it points to the next element in cost.

This is a good time to show you why the type of each variable is important. According to Figure 12-5, a float-type variable uses 4 bytes of memory, so the elements of cost are at intervals of 4 in memory. Since cost is an array of floats, we declared cpntr as a type float pointer. When the compiler translates the cpntr++ statement, it automatically generates an 8086 instruction which adds 4 to the value of cpntr so it points to the next element in cost.

When this first section of the program runs, it will send the "Enter 10 costs..." message to the screen and wait for you to enter a value. After you enter a value and press the space bar or the Enter key, the program will put the value in the array and wait for you to enter the next value.

After all 10 values are read in, the cpntr pointer is reset to point at the start of the cost array with the cpntr = cost; statement, so we can process the 10 values read. To process the 10 values we use a for loop, as in the previous examples. Now let's see how we compute each cost.

In the previous examples we added a fixed profit of 15 to each cost, but this is not very realistic. A more realistic approach is to compute profit as a percentage of the cost and add the computed profit to the initial cost for each item. The statement *ppntr = *cpntr + 0.25*(*cpntr); does this. In more English-like terms it says, "get the cost pointed to by cpntr, multiply that value by 0.25, add the value pointed to by cpntr to the result, and write

```
/* C PROGRAM F12-12.C */
/* float pointers and reading data from the keyboard */

#include<stdio.h>
#define MAX_PRICES 10
void main ()
{
     float cost[MAX_PRICES], prices[MAX_PRICES];
     float *cpntr = cost, *ppntr = prices;
     int i;

     printf("Enter %d costs. After each cost press "
               "space or enter.\n", MAX_PRICES);
     for(i=0; i < MAX_PRICES; i++)
     {
          scanf("%f", cpntr);
          cpntr++;
     }
     cpntr = cost;  /* reset cost pointer to start of array */
     for(i=0; i < MAX_PRICES; i++)
     {
          *ppntr = *cpntr + .25 * (*cpntr);
          printf("cost= %6.2f, price= %6.2f \n",*cpntr,*ppntr);
          cpntr++; ppntr++;
     }
}
```

FIGURE 12-12 Program using float pointers and the scanf function.

the result to the memory locations pointed to by ppntr." Note that the * symbol is used to represent the multiplication operation as well as to represent the "contents of the memory location pointed to by" a pointer. The meaning of a * in a statement is usually clear from how it is used.

After we compute each selling price, we call printf to display the entered costs and the computed prices on the screen. Since we want to print floating-point values, we use %f format specifiers. The 2 between the % and the f indicates that we want the values rounded off to two digits to the right of the decimal point. This is appropriate for money values. The 6 between the % and the f indicates that the values will have a maximum of 6 digits, including the two to the right of the decimal point. This number is optional. Note that we use *cpntr to pass the current cost value to printf and *ppntr to pass the current price value to printf. We then increment the two pointers, cpntr and ppntr, so they point to the next locations in their arrays.

Now that you have some experience with int and float pointers, let's take a look at some char pointers which work just a little bit differently.

CHAR POINTERS AND CHARACTER STRINGS

Some programming languages such as BASIC have a string data type which is used for ASCII code sequences. In C you just use an array of type char to store strings. The last two examples in Figure 12-6 show how to declare char arrays. The next-to-last example in Figure 12-6 shows how you can initialize a char array with a desired string of ASCII codes. Remember that when you initialize a char array in this way, the compiler automatically includes a null character as a sentinel at the end of the string.

As with int and float arrays, the name of a char array is a pointer to the first element in the array, but again, this pointer is a constant, so it cannot be incremented, etc. It is often useful to declare a variable pointer to the start of a char array and use this pointer to access the array. Figure 12-13 shows you how to declare char pointers and some of the different ways to work with character strings in C programs.

At the top of Figure 12-13 note that you can use the #define preprocessor directive to declare a constant string. Whenever the compiler finds the identifier exitmess, it will substitute the constant string "password incorrect." Since this string is a constant, it cannot be modified in the program.

The first char example in Figure 12-13 does several jobs. It declares a char-type pointer called greeting and sets aside 2 bytes of memory to store the pointer. It allocates 14 bytes of memory and initializes these bytes with the ASCII codes for the string "Good Morning" and a null character. Finally, it initializes the pointer, greeting, with the address of the first character in the string.

The printf("%s\n", greeting); statement in main shows how you can get this message printed out on the screen. To let printf know that you are passing it a string, you use the %s format specifier. To identify the string you want to send to the screen, you simply use the name of the pointer to the string. Note that you do not have to put a * in front of the name of the pointer as we did for the float pointer in the printf( ) statement in Figure 12-11. For string operations the compiler assumes that the name of the pointer refers to the whole string. Since the

```
/* examples of declaring and using char type pointers */

#include <stdio.h>
#define exitmess "password incorrect"

void main()
{
char *greeting = "Good morning, ";     /* pointer to type char location, initialized with string */
char wakeup[20] = "Good morning\n";    /* array of 20 char initialized with string shown */
char *message;                         /* declare pointer named message,
                                          but allocate no storage */
char name[20];

printf("%s\n", exitmess);

printf("%s\n", greeting);
printf("%s\n", wakeup);

message = "Hello there.";              /* allocate storage and load string into locations
                                          starting where pointer message points */
printf("The message is, %s\n", message);

printf("Please type in your name and press the Enter key.\n");
gets(name);
printf("%s%s\n", greeting, name);
}
```

FIGURE 12-13 Declaring and using char type pointers.

pointer greeting initially points to the first element in the string, a term such as *greeting would refer only to the first element in the string rather than to the whole string. Don't make this overly complicated in your mind. Just remember that you don't use a * in front of a pointer to a string unless you want to refer to just the first character in the string or individual elements in the string.

The char wakeup[ ]="Good morning.\n"; statement in Figure 12-13 declares an array of characters and initializes the elements of the array with the ASCII codes for the specified string. An ASCII null character, 00H, will automatically be inserted as a sentinel at the end of the string. As we said before, the name of an array is a pointer to the first element in the array, so you can print this message with a statement such as printf("%s \n", wakeup);. Note that here again you do not have to use a * in front of wakeup to tell printf that you want to print the contents of string named wakeup.

The char *message; declaration in Figure 12-13 declares a char-type pointer and sets aside a couple of memory locations for the pointer. However, this statement does not assign any value to the pointer, and it does not allocate any memory for storing a string. When the compiler reads the message="Hello there."; statement in main, it will allocate some memory locations for the string "Hello there." and store the ASCII codes for the string in the allocated memory bytes. The compiler will also initialize the pointer named message with the starting address of the memory allocated for the string. Note again that we referred to the string simply with the name message. The compiler is smart enough to know that message refers to the entire string.

The char name[20]; statement in Figure 12-13 allocates 20 bytes of memory for an array of characters but does not initialize these bytes. The last three statements in main show how you can read a string in from the keyboard and put it in this array.

The printf( ) statement at the start of this section simply prompts the user to enter his or her name and press the Enter key. The second line in this section of the program uses the predefined function gets( ) to read characters entered on the keyboard. You tell gets( ) where to put the characters by passing it a pointer to some char-type locations. In this example, name is a pointer to the array we declared, so we just pass name to gets( ) by including it in the ( ). Gets( ) keeps reading ASCII codes from the keyboard and putting them in the array until it reads the code for a carriage return. When it reads a carriage return, gets( ) puts a null character at the end of the stored string and returns to main. The final printf( ) statement sends the declared string "Good morning," to the screen and then sends the string read in from the keyboard to the screen.

As you look at this last example the question that may occur to you is, Why didn't we use the scanf( ) function that we showed you in Figure 12-12 to read in the string? The answer to this is that scanf terminates when it finds a space, a tab, or a carriage return. Therefore, the space between a first name and a last name would terminate scanf, and only the first name would be read in and put in the array. The scanf function with a %s format specifier works fine if you want to read in only a single character or a single word.

Two important points to remember when working with character arrays or strings are as follows:

1. Use just the name of the array or the name of the pointer to refer to the array. You don't need an * in front of the name unless you want to refer to just the first character in the array or individual elements in the string.
2. You must tell the compiler to allocate storage for a string with a statement such as char name[20]; before you can read in a string from the keyboard. You cannot just declare a pointer with char *message; and then gets(message); because the char *message declaration doesn't allocate any space to put the characters read from the keyboard. It just declares a pointer.

Now that you know how to declare different C data types, how to send messages to the screen, and how to read strings from the keyboard, you should be able to write some simple programs to entertain your friends. To make your programs more interesting, you need some more instructions in your toolbox. In the next sections we show you the different C "instructions" or operators you can use to perform computations, etc., in your programs.

C Operators

THE ASSIGNMENT OPERATOR

The assignment operator in C is simply the = sign. We have already used the assignment operator in the preceding program examples without bothering to give it a name. A statement such as side_a = 3.0;, for example, assigns a value of 3.0 to the variable side_a. This corresponds to an assembly language instruction such as MOV SIDE_A,3.

The = sign says "evaluate the expression to the right of the = and write the result in the variable to the left of the =." The statement prices[index] = cost[index] + 15;, for example, adds 15 to an element from the cost array and puts the result in the corresponding element in the prices array.

In 8086 assembly language you used MOV instructions to copy the contents of one memory location to another. One way to do this in C is with a simple assignment statement. If you have two variables such as maxval and curval of the same type, you can copy the value of curval to maxval with the statement maxval = curval;.

ARITHMETIC OPERATORS

Operation	*Symbol*	*Examples*	
Addition	+	a = c + d;	
Subtraction	−	a = c − d;	
Multiplication	*	a = 4*b	
Division	/	a = c/d;	
Modulus	%	a = c%d;	/* a = remainder of c/d */
Increment	+ +	index + +;	/* increment index by one */
		a = a + b + +;	/* postfix increment */
			/* add b to a, then inc b */
		a = a+ + +b;	/* prefix increment */
			/* inc b, add result to a */
Decrement	− −	count − −;	/* decrement count by one */
		a = a − b − −;	/* postfix decrement */
			/* subtract b from a, decrement b */
		a = a+ − −b;	/* prefix decrement */
			/* decrement b, then add b to a */

BITWISE OPERATORS

These operators correspond to assembly language instructions such as AND, OR, XOR, NOT, ROL, and ROR. As with the assembly language instructions, they perform the specified operation on a bit-by-bit basis. The AND operator, for example, logically ANDs each bit of one operand with the corresponding bit of the other operand. For reference, here are the C bit operators and some examples of each.

Operation	*Symbol*	*Examples*	
AND	&	a = a & b;	/* each bit of b ANDed with corresponding bit in a, result in a */
		a = a & 0xff;	/* mask upper 8 bits of int a */
OR	|	a = a | b;	/* each bit of b ORed with corresponding bit in a, result in a */
		a = a | 0x8000;	/* set MSB of int in a */
XOR		a = a ˆ b;	/* each bit in b is XORed with corresponding bit in a, result in a */
		a = a ˆ 0x000f;	/* invert low nibble int a */
NOT		a = ˜a;	/* invert bits in a, result in a */

Operation	*Symbol*	*Examples*	
Shift-left	<<	a=a << 4;	/* **shift bits in a 4 bit positions left around loop. This corresponds to 8086 assembly language sequence MOV CL, 04H, ROL a, CL.**
Shift-right	>>	a=a >> 8;	/* **shift bits in 8 bit positions right around loop. This corresponds to 8086 assembly language sequence MOV CL,08H, ROR a, CL. It effectively swaps the bytes of a if a is type int.**

COMBINED OPERATORS

It seems that many experienced C programmers have a habit of trying to pack as much action as possible in a single program statement. This often makes the statement somewhat difficult to decipher. As we go through the rest of the book we will try to show you some of the more common shortcuts so that you can use them or at least recognize them when you see them. To start we will show how expressions using the operators in the previous sections are commonly written in shortened form. Again, the best way to do this seems to be with a list of examples that you can easily refer to. Once you see the pattern of these, you will find them quite easy.

Operation	*Standard Form*	*Combined Form*
Addition	a=a+b;	a + =b;
Subtraction	a=a−b;	a − =b;
Multiplication	a=a • b;	a •=b;
Division	a=a/b;	a /= b;
Modulus	a=a%b;	a %=b;
AND	a=a&b;	a &=b;
OR	a=a|b;	a |=b;
XOR	a=a^b;	a ^=b;
Shift-left	a=a<<b;	a <<=b;
Shift-right	a=a>>b;	a >>=b;

RELATIONAL OPERATORS

Relational operators are used in expressions to compare the values of two operands. If the result of the comparison is true, then the value of the expression is 1. If the result of the comparison is false, then the value of the expression is 0. These comparisons are usually used to determine which of two actions to take. You will see many more examples in a later section which discusses how the standard program structures are implemented in C, but a simple example here using the "greater than or equal to" operator should help you see how these are used.

Remember in Chapter 4 we showed you how to implement an algorithm which turned on a light if the temperature in a printed-circuit-board-making machine was equal to or greater than a preset value. To implement this decision we used a compare instruction and a conditional jump instruction. In C you might implement this action with a couple of statements such as:

```
if (current_temp >= run_temp)
{
    heater (off);
    green_light (on);
}
```

We assume here that the value of current_temp was read from an A/D converter by calling an assembly language procedure before the if statement. (Later in the chapter we show you how to do this.) If the value of current_temp is not equal to or greater than the predeclared value of run_temp, then the comparison is false, and the statements in the curly braces will be skipped over. If the expression in ( ) evaluates to true, then the two statements in the curly braces will be executed. In the first of these statements we call a function called heater and pass it a value which will turn the heater off. Likewise, in the second statement we call a function called green_light and pass it a value to turn on the green light. The heater function and the green_light function would most likely call assembly language procedures to manipulate the actual hardware.

Here is a list of the C relational operators. As you read each of these, mentally insert them in a statement such as "if (a = = b) { }," to help you remember how they are used. Note that the = = used here has a very different meaning from the single = used for assignment.

Operator	*Symbol*
Equal to	= =
Not equal to	!=
Greater than	>
Greater than or equal to	>=
Less than	<
Less than or equal to	<=

LOGICAL OPERATORS

In the last section we showed you how the relational operators are used to choose between two actions in, for example, an IF-THEN-ELSE structure. The C logical operators allow you to include two or more conditions in a decision such as this. The three logical operators and the symbols which represent them are as follows.

<table>
<tr><th><em>Operator</em></th><th><em>Symbol</em></th><th><em>Examples</em></th></tr>
<tr><td>AND</td><td>&&</td><td>if(curtemp < maxtemp && curpress < maxpress)
{
green_light(on);
}
/* green light on only if both conditions true */</td></tr>
<tr><td>OR</td><td>||</td><td>if (curtemp > maxtemp || curpress > maxpress)
{
   red_light(on);
}
/* red light on if either condition met */</td></tr>
<tr><td>NOT</td><td>!</td><td>if (!a)
{
  statements;
}
/* do statements if a is false (= 0)
   skip over statements if a is true (= 1) */</td></tr>
</table>

OPERATOR PRECEDENCE

In the preceding sections we have shown you most of the C operators. We will show you the few remaining operators in later program examples where they may make more sense. The next topic we have to discuss here is the priority or precedence of the C operators. To properly evaluate or write an expression which has several operators, you have to know the order in which the operations are done. As an example of this, in the statement *prices = *cost + 0.25*(*cost);, how did we know that the 0.25 would first be multiplied by *cost and then the result added to *cost? The answer to this is that the multiplication operator has a higher priority or precedence than the addition operator, so the multiplication gets done before the addition.

As another simple example of this, suppose you have an expression such as a/b + c/d. From ordinary algebra you know that division also has a higher precedence than addition, so the two divisions will be done first, and then the results of the two divisions will be added together.

Shown below in descending order is the precedence of the C operators. For reference we have included some operators that we haven't discussed yet, so don't worry if you don't recognize all of these. To help you identify the different operators, we have included simple examples of each. In the paragraphs following this list we show you some more examples to help stick the important ones in your mind.

NOTE: All the operators in a group have the same priority.

Operator	*Example*	
( )	4*(9+2)	/* operation in parentheses done first */
[ ]	cost [3]	/* fourth element in array cost */
.	class.ssnmbr	/* pointer to ssnmbr member of structure */
−>		/* indirect structure operator */
−	a= −23;	/* negation */
+	a= +28;	/* positive value */
¯	a= ¯a ;	/* invert each bit in a */
*	*cpntr	/* contents of location pointed to by cpntr */
&	&headcount	/* address of headcount */
+ +	index + +	/* increment operator */
− −	count − −	/* decrement operator */
sizeof	count = sizeof cost;	/* determine # of bytes in cost */
*	a * b	/* multiplication */
/	a/b	/* division */
%	a%b	/* modulus-remainder from division */
+	a + b	/* addition */
−	a − b	/* subtraction */
<<	a<<4	/* shift bits of a left 4 bit positions */
>>	a>>8	/* shift bits of a right 8 bit positions */

Operator	*Example*	
<	if (a<10)	/* less than */
>	if (temp>30)	/* greater than */
<=	if (a<=10)	/* less than or equal to */
>=	if (temp>=5)	/* greater than or equal to */
==	if (a==b)	/* relational equal */
!=	if (a!=b)	/* relational not equal */
&	a & 0xfff0	/*AND a with fff0H to mask lowest nibble*/
|	a | 0x8000	/* OR a with 8000H to set MSB */
^	a ^ 0x000f	/* XOR a with 000fH to invert 4 LSBs
&&	If (condition 1 && condition 2)	/* both 1 AND 2 */
||	If (condition 1 || condition 2)	/* 1 OR 2 */
simple assignment		
=	a=4;	/* simple assignment */
combined assignment (see previous examples)		
*= /= %= += -= <<= >>= &= |= ^=		

As we showed before, the precedence of C arithmetic operators is basically the same as in ordinary algebra, so you should have little trouble with these. In an expression such as 3+4*a, the multiplication will be done before the addition, because multiplication has a higher precedence than addition. If you want the addition to be done first, you can write the expression as (3+4)*a. Parentheses have a higher precedence than multiplication, so any operations within parentheses will be done first. If there is any possibility of misinterpreting an expression, you should use parentheses to make it clear.

The only case where you may initially need a little help to understand the precedence of operators is with the increment and decrement operators, so we will discuss these.

If you use the increment operator, ++, in a simple statement such as index++;, you can write the ++ after index or in front of it. In other words, the statement ++index; and the statement index++; will each increment the value of index by 1. When ++ or -- is used in more complex expressions, however, the placement of the operator is important.

In a statement such as Y=(a+ ++b)/10;, for example, the value of b will first be incremented by 1 and the result added to a. The sum of a and the incremented b is then divided by 10 and the result assigned (copied) to the variable y. Incrementing or decrementing a variable before it is used in the expression is often referred to as a *prefix* operation.

If you write the statement as Y=(a+ b++)/10;, the current value of b will be added to a. Next the result of this addition will be divided by 10 and assigned to Y. Finally, the value of b will be incremented by 1. Using a variable and then incrementing or decrementing it is often referred to as a *postfix* operation.

The simple rules here then are: Put the ++ or -- operator in front of the variable name if you want the variable incremented or decremented before it is used to evaluate the expression. Put the ++ or -- operator after the variable name if you want the current value of the variable used to evaluate the expression.

Statements such as those shown in the preceding paragraphs are usually quite straightforward, once you understand the prefix and postfix concept. Another situation where you will often see the increment and decrement operators is in conditional expressions such as while (a++ <20), which might be used at the start of a WHILE-DO structure. The ++ is after the variable a, so you know that the current value of a is used to evaluate the expression, and then a is incremented. The expression then says "compare the current value of a to 20 and then increment a." If the value of a is less than 20, do the statements following the while.

To see if you understand how this works, try interpreting the statement while (--b >0) { }. The -- is before the variable b, so b will be decremented and the decremented value of b compared with 0. If the decremented value is greater than 0, the statements following the while will be executed. If the decremented value of b is equal to 0, execution will go to the next statement in the program after the while block.

Throughout the preceding discussions we have given you little glimpses of how the standard programming structures are implemented in C. In the next section we take a closer look at these.

Implementing Standard Program Structures in C

As we tried to show you in Chapter 3, the most successful way to write any program is to solve the problem mentally; write the algorithm for the solution using the basic IF-THEN-ELSE, CASE, REPEAT-UNTIL, WHILE-DO, and FOR-DO structures shown in Figure 3-3; and finally translate the algorithm to an appropriate programming language. The C implementation of these structures is very close to the pseudocode for them, so the translation is usually quite easy. In this section we discuss each of these and show you some more C programming techniques.

IF-THEN AND IF-THEN-ELSE IMPLEMENTATION

The general format of the IF-THEN-ELSE structure in C is:

```
if  (condition)
{
     statement;
     statement;
}
else
{
     statement;
     statement;
}
```

Condition in this format represents some expression such as currtemp = = maxtemp. If the condition expression evaluates to 1 or any nonzero value, the block of statements under the if will be executed. If the condition expression evaluates to 0, the block of statements under the else will be executed. The else block can be omitted if you want just an IF-THEN instead of an IF-THEN-ELSE. Note that the curly braces are not needed for the case where the if block contains only one statement. Likewise, the curly braces are not needed in the else block if it contains only one statement.

The program section in Figure 12-14 shows a simple IF-ELSE structure and introduces you to getch( ), another predefined function which you will probably want to use in your programs. This example also gives you a little more practice with operator precedence.

At the start of the program we declare a char-type variable and give it the traditional name ch. After printing a couple of prompt messages, we use an IF-ELSE to determine a course of action based on the user's response. To evaluate an expression such as the if condition in Figure 12-14, you start with the innermost parentheses and work your way out. The getch( ) part of the if expression calls the predefined function getch( ). The getch( ) function sits in a loop until the user presses a key on the keyboard. When the user presses a key, getch( ) terminates and returns the ASCII code for the key pressed. In this example the ch = getch( ) means that the returned ASCII value will be assigned (copied) to the variable named ch. This completes the action in the inner parentheses. The value produced by these actions is the ASCII code stored in ch.

The = = 'n' next in the expression compares the value in ch with the ASCII code for a lowercase n. If the values are the same, the entire expression is true (evaluates to 1) and the statements in the if block will be executed. If the value in ch is not equal to the value of the ASCII code for a lowercase n, the || ch = = 'N' part of the expression compares the value of ch with the ASCII code for an uppercase N. Remember that the || symbol represents the logical OR operation, so the overall expression is true if ch = n OR ch = N. If the entered character was an N, the statements in the if block will be executed. If the character was not an n or an N, the entire expression evaluates to 0, and statements in the else block will be executed.

> NOTE: The expression for the if statement is evaluated from left to right, so the (ch = getch( )) is done first and the result compared with n. For the second comparison you just write ch = 'N', because ch already has the value read in from the keyboard. If we had used (ch = getch( )) = = 'N' here, execution would sit in getch( ) until the user pressed another key! Incidentally, if you want the key pressed by the user to be echoed to the CRT, you can use the getche( ) function instead of the getch( ) function.

The exit( ); statement in the if block calls a predefined function which terminates the program and returns control to the operating system (DOS prompt).

In the else block we display a message to let the user know that something is happening, then we use the "goto start" statement to send execution to the beginning of the program. The goto statement in C corresponds to the unconditional JMP instruction in assembly lan-

```
/* C PROGRAM F12-14.C */
#include <stdio.h>
void main()
{
  char ch;
  start:      printf("Game over.\n");
              printf("Enter y to play another game,. n to quit.\n");
        if ((ch = getch()) == 'N' || ch == 'n')
              {
              printf("Goodbye.\n");
              exit();
              }
        else
              {
              printf("Here we go again.\n");
              goto start;
              }
}
```

FIGURE 12-14 Basic if-else example.

guage. As in assembly language, the name start represents a label which you place in front of the instruction statement that you want execution to go to. In C you write a : after the label, just as you do in assembly language. In this example we put the start label next to the first printf statement, just to show you how to write labels.

> NOTE: The label for a goto must be in the same function as the goto statement.

Some structured programming fanatics say that you should never use even a single goto in a program. This attitude is probably a reaction to the way goto statements were abused in old BASIC programs. To us, however, using a simple goto to rerun the entire program is the clearest way to do it. In reality, even if you hide the action in some other structure, the compiler will usually generate an unconditional jump instruction to implement the action.

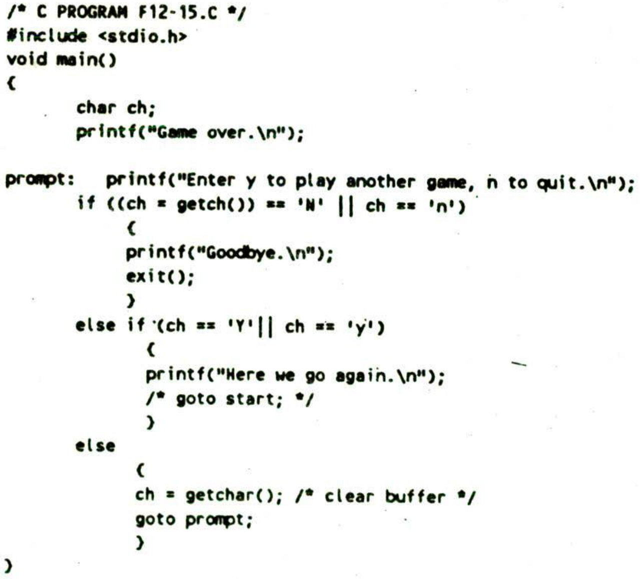
The program fragment in Figure 12-14 has a minor problem. It thoroughly tests to see if the user entered an n or an N and exits if either of these was entered. However, if any other key is pressed, the else-block statements start the game over again. Figure 12-15 shows how you can use a nested if-else structure to provide three alternative actions based on the key pressed. For an n or N the statements in the first if block will be executed. For a y or a Y the statements in the second if block will be executed. For any other key the statements after the final else will be executed. In a later example we show you how a "real C programmer" might write this program segment to avoid the direct goto statement in the final else block.

MULTIPLE CHOICES—THE SWITCH STATEMENT

To implement algorithms with more than three choices, you can nest additional if-else sections, but often a more efficient way to do this is with the *switch* structure. The C switch structure is essentially the same as the CASE structure we showed you in Figure 3-3. The general format for the switch statement is:

```
switch (variable)
        {
        case value1:
        {
        statements;
        break;
        }
        case value2: statement(s); break;
        case value3: statement(s); break;
        default: statement;              /* optional */
```

Variable in this statement must be some quantity such as an int or char which can be evaluated as an integer. Value1 in the first case line represents some value of the variable used to make the decision. After each case line you write the statement(s) you want executed if the variable has that value. If, for example, the value of variable is equal to value1, the statements after case value1: will be executed. The break statement at the end of this block of statements will cause execution to skip over the rest of the choices in the structure. If you leave out the break statement, the actions for the next case after the selected case will be executed. The optional default directive at the end of the switch structure allows you to specify the action(s) you want taken if the value of variable does not match any of the specified values.



```
/* C PROGRAM F12-15.C */
#include <stdio.h>
void main()
{
        char ch;
        printf("Game over.\n");

prompt:   printf("Enter y to play another game, n to quit.\n");
        if ((ch = getch()) == 'N' || ch == 'n')
             {
             printf("Goodbye.\n");
             exit();
             }
        else if (ch == 'Y'|| ch == 'y')
               {
               printf("Here we go again.\n");
               /* goto start; */
               }
        else
             {
             ch = getchar(); /* clear buffer */
             goto prompt;
             }
}
```

FIGURE 12-15 Nested if-else example.

Figure 12-16 shows how you might use the switch statement to implement a "command recognizer" in one of your programs. This example is modeled after the commands available at the highest menu level in the Borland TC development environment we discussed earlier in the chapter. To get to the main menu in TC, you press the F10 key. To select the desired submenu you then press the key which corresponds to the first letter in the name of the submenu. The choices are F, E, R, C, P, O, D, and B. Each of these options brings up a lower-level menu or carries out a command.

In the program in Figure 12-16 we use our new friend getch( ) to read a character from the keyboard. We then use a switch structure to evaluate the character and decide what action to take. To simplify the basic structure of this example, we call a function to implement each of the desired actions. Actually, for this example we show the function calls as comments, because we did not want to declare and define all these functions. When execution returns from the called function, the break statement at the end of that line will cause execution to skip to the next statement after the switch structure. If the key pressed by the user does not match any of the choices, the default: edit_window( ); statement at the end of the block sends execution back to the edit operation. You can have only one value in each case evaluation, so if you want the program to accept lower- or uppercase letters, you have to put case lines in for each. The line case 'F': followed by the line case 'f': file( );break;, for example, will call the file function if the user enters either a lower- or uppercase f. A more versatile alternative is to write a small function which converts all entered characters to lowercase before entering the switch structure. We leave this for you to do as an exercise at the end of the chapter.

THE WHILE AND DO-WHILE IMPLEMENTATIONS

In Chapter 3 we showed you how the WHILE-DO and the REPEAT-UNTIL structures are used to loop through a series of statements. In C these two structures are called the while and the do-while, respectively. The major difference between the two structures is when the exit test is done. For comparison, Figure 12-17 shows how the two are implemented in C.

As you can see, in the while loop in Figure 12-17*a*, the condition is evaluated before any statements are executed. If the condition expression initially evaluates to 0, execution will simply bypass the block of statements under while and go on with the rest of the program. In this case none of the statements in the while block will be executed. If the condition expression initially evaluates to a nonzero value, the statements in the while block will be executed once. Then the condition expression will be evaluated again, and if the result of the evaluation is still nonzero, the statements in the while block will be executed again. Looping will continue until the condition expression evaluates to 0.

The key point of a while loop is that the condition is tested before any statements are executed. In most cases this "look before you leap" approach is the best one, and most loop algorithms can be written in this way.

For those cases where you want the loop statements to be executed once before the condition is checked, C has the do-while structure shown in Figure 12-17*b*. In this structure the statements in the do-while block are executed once, and then the specified condition expression is evaluated. If the condition expression evaluates to 0, the do-while terminates and execution goes on to the rest of the program. With this structure, then, the statements in the do-while block will always be executed at least once. If the condition expression evaluates to a nonzero value after executing these statements, the statements in the do-while block will be executed again and the condition expression evaluated again. Looping will continue until the condition expression evaluates to 0. Note that in this structure there is a semicolon at the end of the while line.

Figure 12-18*a* shows how you can use a while loop to make a user enter a Y or an N in response to a prompt. This approach avoids using a goto such as the goto prompt; statement in Figure 12-15. The declaration statement for ch at the start of the program gives it a null value, so the first time the condition for the while

```
#include<stdio.h>
void main()
{
 char ch;
 ch=getchar();
 switch (ch) {
 case 'F':
 case 'f': /* file_menu();      */ break;
 case 'e': /* edit_window();    */ break;
 case 'r': /* run_menu();       */ break;
 case 'c': /* compile_menu();   */ break;
 case 'p': /* project_menu();   */ break;
 case 'o': /* options_menu();   */ break;
 case 'd': /* debug_menu();     */ break;
 case 'b': /* break_menu();     */ break;
 default : /* edit_window();    */ ;
          }
}
```

FIGURE 12-16 Example of C switch structure.

```
/* while format */

while(condition)
             {
             statement(s);
             }
```

(*a*)

```
/* do-while format */
do
             {
             statement(s);
             }
while(condition);
```

(*b*)

FIGURE 12-17 (a) Basic format of C while structure. (*b*) Basic format of C do-while structure.

```
/* while example */
#include <stdio.h>
void main()
{
char ch = 0x00;  /* assign initial value to ch */
while(ch!='n'&& ch!='N'&& ch!='y'&& ch!='Y')
     {
     printf("Enter y to play another game, n to quit.\n");
     ch=getch();
     }
if (ch=='n'|| ch=='N')
     {
     printf("Goodbye.\n");
     exit();
     }
else
     {
     printf("Here we go again.\n");
     /* goto start */
     }
}
```

(*a*)

```
/* do-while example */
#include <stdio.h>
void main()
{
     char ch;
     do
     {
     printf("Enter y to play another game, n to quit.\n");
     ch=getch();
     }
while(ch!='n'&& ch!='N'&& ch!='y'&& ch!='Y');
if (ch=='n'|| ch=='N')
     {
     printf("Goodbye.\n");
     exit();
     }
else
     {
     printf("Here we go again.\n");
     /* goto start */
     }
}
```

(*b*)

FIGURE 12-18 (*a*) Example of C while structure. (*b*) Example of C do-while structure.

statement is tested, the result is false. Therefore, the ch = getch( ); statement part of the while is executed. When getch( ) returns a new value to ch, the condition expression for the while will be checked again. Execution will stay in this while loop until getch( ) returns a y, Y, n, or N. After it exits the loop, execution goes to the if-else section of the program to determine the actions to take based on the value returned by getch( ) and assigned to ch.

Figure 12-18*b* shows how the same program section can be implemented as a do-while. In this example we did not need to give ch an initial value, because the ch = getch( ); statement at the start of the do-while gives ch a value before any tests are made. The ch = getch( ) statement will be repeated until the value of ch matches one of the values in the condition test part of the do-while.

It is not obvious in the examples shown in Figure 12-18, but in most cases the while structure is a better choice than the do-while, because the condition is checked before any action is done.

THE FOR LOOP

As we showed you in several previous program examples, a for loop can be used to do a sequence of statements a specified number of times. The general format of a for loop is:

```
for (initialization(s); test; modify)
{
statement(s)
}
```

To refresh your memory, Figure 12-19 shows a simple example of a for loop. The initialization in this example assigns a value of 0 to the variable count. If you want to, you can include more than one initialization here. You might, for example, include two initialization statements such as count=0; b=23; to initialize a variable called b with a value of 23 as well as initialize the loop variable count.

The test part of this example compares the value of count with the terminal value. If the value of count is not equal to the terminal value, the statements in the loop will be repeated.

The count-- in our example represents the "modify" part of the for. This is where you specify what you want to change each time around the loop so that the loop eventually terminates. In some C programs you may see more than one action statement in the modify section of the for( ). You might, for example, see something such as "count++, index=index+4;" in the modify section. These two statements will increment count by 1 and increment index by 4 each time through the loop. Our personal feeling is that the program is more readable if you put only the loop variable initialization and loop variable modification in the for parentheses.

To give you a little more challenging example of a for loop and teach you more about arrays, the first part of the program in Figure 12-20*a* shows how you can use nested for loops to read maximum and minimum temperature values from the keyboard and put the values in a two-dimensional array. The last section of the program uses another for loop to compute the average temperature for each day and display all the results.

The int temps[7][3]; statement at the start of the program declares an array of seven rows and three columns. To help you visualize this, Figure 12-20*b* shows the array in diagram form. As you can see, there is one row for each of the 7 days of the week. Also, there is one column for the daily maximum temperatures, one column for the daily minimum temperatures, and one column for the averages that will be calculated. The arrow looping through the array shows the sequence that the array values are stored in memory. As you can see, the three elements in the first row are stored in the three lowest memory locations, the three elements in the next row are stored in the next three memory locations, etc.

```
#include<stdio.h>
int count;
void main()
{
            int count;
            for (count=10; count>0; count--)
            {
            printf("%d\n ", count);
            }
            printf("blastoff!");
}
```

FIGURE 12-19 Example of simple count-down for loop.

The elements of the array are stored in sequence in memory, so you could access the elements in this array as if it were a one-dimensional array of 21 elements. In other words, you could set up a pointer to the first element in the array and then keep incrementing the pointer to access the other elements in the array. The problem with this method is that you lose the row and column information.

A much more versatile way to access the elements in this array is with row and column index values. The index for an array starts from zero, so the term temps[0][0] is a way to refer to the element in the first row of the first column. Likewise, the term temps[0][1] is a way to refer to the element in the second column of the first row, and the term temps[6][2] is a way to refer to the value of the third element of the seventh row.

In the program in Figure 12-20*a* we use the variable *i* to index a desired row and the variable *j* to index a desired column in the array. The inner for loop in the program uses *j* to access the elements in a row. The first time the inner loop executes it will put the value returned by scanf in the first element in the row. The second time the inner for loop executes, it will put the value returned by scanf in the second element in the row. Since the inner loop is set to terminate for *j* < 2, the inner loop will then terminate and execution will go back to the outer for loop.

The outer for loop uses *i* to access the desired row in the array. The first time through the outer loop *i* = 0, so the first row in the array will be accessed. The next time through the loop *i* has been incremented to 1, so the second row in the array will be accessed. This process is essentially the same as the nested delay loops that you met in earlier chapters.

The scanf function requires that you pass it a format specifier to tell it what type of data it will be reading and that you pass it the address of the location where you want the data put. You use the %d specifier to indicate that you want the data treated as a decimal value, and you use the term &temps[i][j] to pass the address of the desired element in the array to scanf. Remember that temps[i][j] is a way to refer to the value of an element in the array, so &temps[i][j] is a simple way to refer to the address of that element. Note that we used *i* + 1 for the value of the day instead of just *i*. An array index starts from zero, but we want the days to be numbered 1 through 7.

After all fourteen temperature values are read in and put in the appropriate locations in the array, we use a single for loop to compute the average temperature for each day and put the computed results in the appropriate row of the third column in the array. The temps[i][j+2]=(temps[i][j]+temps[i][j+1])/2 statement shows how you can add a constant to the *j* index value to access the different elements in a row. Likewise, in the last printf statement in Figure 12-20*a*, we add constants to the *j* index to access the three elements in a row. Textbooks often refer to this as "pointer arithmetic."

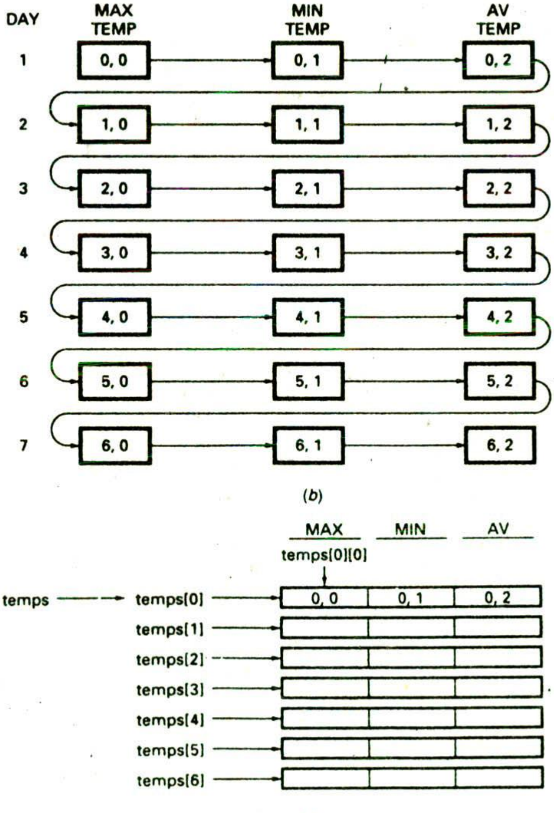
Now that you know the array-index method of ac-

```
/* C PROGRAM F12-20A.C */
/*Program to read max and min temperatures, then compute average */

#include <stdio.h>
int temps[7][3];              /* extern so other modules can access */

void main()
{
     int i, j;
     for (i=0; i<7; i++)
         {
           printf("Enter max temp for day %d,"
               "then min temp.\n", i+1);
           for(j=0; j<2; j++) /* read max, then min */
          scanf ("%d", &temps[i][j]);
         }
     j=0;  /* reset column index */
     for(i=0; i<7; i++)
     {
     temps[i][j+2] = (temps[i][j] + temps[i][j+1])/2;
     printf("For day %d max = %d min = %d av = %d \n",
            (i+1), temps[i][j], temps[i][j+1], temps[i][j+2]);
     }
}
```

(a)



(b)

(c)

FIGURE 12-20 (a) Program showing index method of accessing elements in two-dimensional array. (b) Two-dimensional array of 7 rows and 3 columns used to store maximum, minimum, and average temperatures for 7 days. NOTE: Arrow shows order that values are stored in memory, going from lowest to highest memory address. (c) Two-dimensional array shown as 7-element array of one-dimensional 3-element arrays. (*See also next page.*)

```
/* C PROGRAM F12-20D.C */
/*Program to read max and min temperatures, then compute average */

#include <stdio.h>
int temps[7][3];
void main()
{
    int i, j;
    for (i=0; i<7; i++)                           /* read values entered */
      {
        printf("Enter max temp for day %d,"
            "then min temp.\n", i+1);
        for(j=0; j<2; j++)                          /* read max, then min */
       scanf ("%d", (*(temps+i)+j));
      }
    for(i=0; i<7; i++)             /* compute averages and print all values */
    {
    *(*(temps+i)+2) = (*(*(temps+i)+0) + *(*(temps+i)+1))/2;
    printr("For day %d max = %d min = %d av = %d \n",
      (i+1), *(*(temps+i)+j), *(*(temps+i)+1), *(*(temps+i)+2));
    }
}
```

(*d*)

```
For day 1 max = 98 min = 68 av = 83
For day 2 max = 89 min = 65 av = 77
For day 3 max = 87 min = 59 av = 73
For day 4 max = 90 min = 67 av = 78
For day 5 max = 86 min = 58 av = 72
For day 6 max = 78 min = 68 av = 73
For day 7 max = 83 min = 69 av = 76
```

(e)

FIGURE 12-20 (*Continued*) (*d*) Program in Figure 12-20a rewritten using pointer notation. (e) Results produced by program in 12-20a or 12-20*d*.

cessing the elements in a two-dimensional array such as this, we will briefly show you the direct pointer method, which is very commonly used by experienced C programmers. Even if you don't choose to use this pointer method yourself, you should understand it well enough to follow it in other peoples' programs.

As we said before, one way of thinking about the array temps[7][3] is as a two-dimensional array with seven rows and three columns. Another common way of thinking of the array named temps is as seven one-dimensional arrays of three elements each. In this view shown in Figure 12-20c, temps[0] is the name of the first three-element array, temps[1] is the name of the second three-element array, and temps[6] is the name of the last three-element array.

The key to understanding how you work with this form is to remember that *the name of an array is a pointer to the first element in the array*. The name temps then is a pointer to the first element in the array of arrays. In this view the first element in the array is the subarray temps[0], so temps is a pointer to temps[0]. One way to represent this relationship in C syntax is temps = &temps[0]. The other way to represent this relationship is *temps = temp[0].

Now, temp[0] is the name of an array of three ints, so temp[0] is a pointer to the first element in the array temps[0]. You can refer to the value of the first element in temps[0] with the expression *temps[0]. This expression simply says "the value pointed to by the pointer temps[0]." In the last paragraph we showed you that *temps = temps[0], so with a little substitution the expression *temps[0] can be written as **temps. The **temps expression, which is the pointer form we wanted to get to, means "the contents of the memory location pointed to by the contents of the memory location pointed to by temps." This is easier to understand if you mentally put parentheses around *temps and think of it as a pointer to the first subarray, temps[0].

The result of all this is that the three equivalent ways to refer to the value of the first element in the first subarray of temps are:

temps[0][0] = *temps[0] = **temps

The expression temps[0][0] is the two-dimensional array method we showed you in Figure 12-20*a*. The expression *temps[0] takes advantage of the fact that temps[0] is a pointer to the first subarray and *temps[0] represents the value pointed to. The expression **temps is just an indirect way to point to temps[0] and then to the value pointed to by temps[0]. The two-dimensional-array form is probably the most intuitive, but most compilers convert it to the pointer form to produce the actual machine code. Therefore, many programmers write array expressions directly in the pointer form.

If you follow that **temps is a valid way to refer to the first element in the first subarray or row of temps, the question that may occur to you is, How do you access the other elements in the array using the pointer form? The answer to this question is that you add index values to the pointer to access the desired element. If you use *i* as the row or subarray index and *j* as the column index as we did in Figure 12-20*a*, then

```
temps[i][j]=*(*(temps+i)+j)
```

The *(temps+i) in the second expression points to the desired subarray. Adding *j* to this changes the value of the pointer to point to the desired element in the subarray. For reference, Figure 12-20*d* shows how the program in Figure 12-20*a* can be written using the pointer notation we have just shown you. If you work your way through this example, you should be well on your way to understanding C pointers. Note that we used the numbers 0, 1, and 2 to index the desired column in the statement which computes the average and in the printf statement. The +0 is not needed in the second term, but we included it to emphasize the position of the column index in the term. Figure 12-20*e* shows the results produced by either the program in 12-20*a* or the one in 12-20*d*.

C Functions

DECLARING, DEFINING, AND CALLING C FUNCTIONS

As we have told you many times before, often the best way to write a large program is to break it down into manageable modules and write each module as a tion or a series of functions. The C functions we used in the preceding program examples are all "predefined." The code for these functions is contained in library files. All you have to do to use one of these functions is to put #include<> at the start of your program to tell the compiler the name of the file which contains the prototype of the function, call the function by name, and in some cases pass some parameters to the function. Now we need to show you how to write and use your own C functions.

To create and use a function in a program, you must declare the function, define the function, and call the function. Figure 12-21*a*, page 418, shows a template or model of how you do each of these, and Figure 12-21*b* shows a simple program example. To help you understand the terms in the templates, we suggest that you look at the corresponding parts in the example program as we discuss the templates. Don't worry about the details of the example program, because after we work through the templates we will discuss the example program more thoroughly. The three templates in Figure 12-21*a* are shown in the order that they appear in programs, but we will discuss them in the order that you usually construct them as you write a program.

The first step in writing a function is to define the actual function. Functions are always defined outside of main( ), because you cannot define one function within another. To actually write the function you will probably work from the inside out. In other words, you will probably first write the data declarations and the action statements which implement the algorithm for the body of the function. Note that the statement block for the function is enclosed in curly braces. After you write the body of the function, you can then decide what values have to be passed to the function and what value, if any, will be returned from the function to the calling program. When you arrive at these decisions you can write the header for the function.

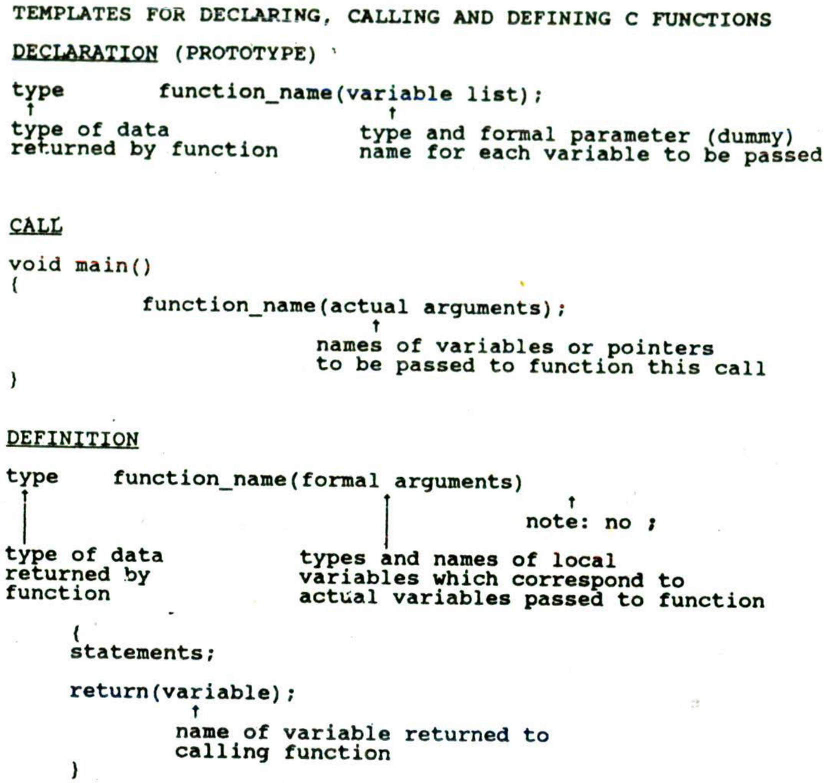
As shown in Figure 12-21*a*, the function header starts with a type such as int, float, char, etc. The type in this case represents the type of the variable returned from the function to the calling program. A C function can return the value of only one variable to the calling program. If the function does not directly return a value to the calling program, you give the function a type void.

> NOTE: Most programmers don't bother to assign a type to the main function, but the Turbo C++ compiler will give a warning if no type is given. You can either make main type void or ignore the warning.

After the function name, you enclose in parentheses the type and name for each function variable that will receive values passed from the calling program. These variables declared in the function header are often called *formal arguments* or *formal parameters*. The trick here is that you usually use different names for particular variables in the calling program and in the function. This makes the function "generic," because you can then pass any variables of the same types to the function in place of the "local" variables declared in the function definition header. Later, when we discuss the details of the example program in Figure 12-21*b*, you will better see how this works.

As an example of a function header, the function header int c2f(int c) in Figure 12-21*b* declares a function called c2f which returns an int value and requires an int value to be passed to it. The int value passed to the function will be automatically assigned to the int variable called c in the function. Also in Figure 12-21*b* the function header void get_temp(int *ptr) defines a function called get_temp which does not return a value, but requires that a pointer to an int type variable be passed to it. Note that function header lines do not have semicolons after them.

After you write the function definition, the next step is to declare the function by writing a *prototype* for the function. This declaration is equivalent to declaring a variable at the start of your program. Note in Figure 12-21*a* that the function prototype declaration at the start of the program has the same format as the function definition header, but it is followed by a ;. This prototype lets the compiler know the name of the function and the types of data to be passed to the function. The compiler uses this information to make sure that the correct data



```
TEMPLATES FOR DECLARING, CALLING AND DEFINING C FUNCTIONS

DECLARATION (PROTOTYPE)

type          function_name(variable list);
 ↑                                ↑
type of data                     type and formal parameter (dummy)
returned by function             name for each variable to be passed


CALL

void main()
{
            function_name(actual arguments);
                                 ↑
                          names of variables or pointers
                          to be passed to function this call
}

DEFINITION

type     function_name(formal arguments)
 ↑                           ↑                  ↑
 |                           |             note: no ;
 |                           |
type of data             types and names of local
returned by              variables which correspond to
function                 actual variables passed to function

     {
     statements;

     return(variable);
              ↑
          name of variable returned to
          calling function
     }
```

(a)

FIGURE 12-21 Declaring, calling, and defining C functions. (*a*) Template. (*See also next page.*)

types are passed to the function when it is called. In large programs the function prototypes are put in a separate header file and pulled into the program at compile time with a #include<> directive. This reduces the "clutter" at the start of the main program.

As shown in the CALL section of Figure 12-21*a*, you call a function with its name and a set of parentheses which enclose the name(s) of the variables being passed to the function. If no variables are passed to the function, you put the term void in the parentheses after the function name.

The variables named in the function call are commonly called *actual arguments* or *actual parameters*. Remember from a previous discussion that when you pass a variable to a function in C, you pass just the value of the variable, or—in other words—just a copy of the variable. If you want the function to be able to access and change the actual value of a variable, you must pass the function a pointer to the variable. Now that you have an overview of the three tasks, let's take a little closer look at the example program in Figure 12-21*b*.

In this example program we first declare an int variable named tempc which will hold the value of a Celsius temperature entered by the user and an int variable called tempf which will hold the value of a Fahrenheit temperature calculated by a function in the program.

The int c2f(int c); statement next in the program is the function prototype declaration for the c2f function. As you should be able to tell from the statement, the c2f function returns an int value and expects to receive a single int value from the calling program. Before we look at the next function prototype, let's work our way through the call and execution of the c2f function.

We call the c2f function with the statement tempf= c2f(tempc); statement. This statement will pass the value of tempc to the function and assign the value returned by the function to tempf. This second effect is the same as you met earlier in statements such as ch=getch( ).

Note that the variable name tempc does not appear in the c2f function block. As we said before, the actual argument passed in the function call is given to the corresponding formal argument identified in the function header. In this case the only formal argument in the header is c, so the value of the actual argument tempc will be assigned to the variable c in the function. In a case where several arguments are being passed to the function, each actual argument will be assigned to the corresponding numbered formal argument.

In the c2f function we declare an additional int variable named f and then we use a familiar formula to calculate the equivalent Fahrenheit temperature for the Celsius

```
/* Declaring, calling, and defining functions */

#include<stdio.h>

int tempc, tempf;          /* external (global) variables */
int c2f(int c);            /* declare function c2f which returns an int value */

void get_temp(int *ptr); /* declare function which modifies a value
                            pointed to, but does not directly return a value */
void main()
{
     get_temp(&tempc);       /* call function get_temp.
                                get_temp writes directly to tempc */
     tempf = c2f(tempc);     /* call c2f function, pass value
                                of tempc to function. Returned value
                                assigned to tempf */

     printf("The temperature in Celsius is %d\n", tempc);
     printf("The temperature in Fahrenheit is %d\n",tempf);
}  /* end of main */

int c2f(int c)               /* define function c2f. Note no ; at end */
     {
     int f;                  /* automatic (local) variable */
     f = 9*c/5 + 32;
     return (f);
     }

void get_temp(int *ptr)      /* define function get_temp */
     {
     printf("Please enter the Celsius temperature.\n");
     scanf("%d",ptr);
     }
```

(*b*)

FIGURE 12-21 (*Continued*) (*b*) Examples in a program.

value passed to the function. The operator precedence rules we showed you earlier in the chapter tell you that c will first be multiplied by 9 and the result divided by 5. Then 32 will be added to the quotient and the result assigned to the variable f. The return(f); statement at the end of the function returns execution to the calling program and passes back the value of f. As we said before, this value is assigned to tempf in the calling program. Incidentally, the parentheses after the return statement can contain any expression which evaluates to an int. You could, for example, write the return statement as return(9*c/5 +32);. For your first programs, however, it is probably better to keep the action "spread out" as we did in the example so you can follow it more easily. Now let's work through the second function in Figure 12-21*b*.

The void get_temp(int *ptr); prototype declaration tells you that the function get_temp does not directly return a value and that the function expects to receive a pointer to an int type variable when called. We call the function with the statement get_temp(&tempc), so the address of the variable tempc is passed to the function. In the get_temp function header, we declared a pointer named ptr with the (int *ptr) after the function name, so the address of tempc will be assigned to ptr when it is passed to the function. In other words, ptr = &tempc.

In the get_temp function we send a prompt message to the user and then use scanf to read the user's response. As you may remember from previous examples, the predefined scanf function requires a format specifier and a pointer to the location where you want it to put the data read from the keyboard. In this call to scanf we pass ptr to it, so the result read from the keyboard will be written to the location pointed to by ptr. Since ptr = &tempc, the value read from the keyboard will be written to tempc. This function has no return statement, because no value is returned to the calling program, but when the scanf("%d",ptr) call is finished, execution will return to the calling program.

Now that you know more about C functions, we need to talk again about the difference between variables declared in a function and variables declared outside any function.

EXTERN, AUTOMATIC, STATIC, AND REGISTER STORAGE CLASSES

Any variable or function declared in a program has two properties, which are sometimes referred to as lifetime and visibility or scope. These terms are best explained by some examples. As we mentioned in an earlier section, variables declared outside of main are by default *extern*.

or—in other words—global. This means that they are visible to or accessible from anywhere in the source file where they are defined or from other files which will be linked with that file. Extern variables are created in memory when the program is loaded and remain there or "live" as long as the program is running. In Figure 12-21*b* tempc and tempf are examples of variables which are extern by default.

We also mentioned earlier that variables declared in a function are by default *automatic*. An automatic variable is "local," which means that it is accessible or visible only within the function where it is declared. Each time you call a function which contains an automatic variable, a temporary storage space is allocated on the stack for that variable. When the function returns execution to the calling program, this storage space is deallocated. An automatic variable then only lives during the execution of the function block where it is declared. In Figure 12-21*b* the variables ptr, c, and f are examples of automatic variables.

Now, suppose that you want to declare a variable within a function so the whole world can't access it, but you want the variable to keep its value from one call of the function to the next. You can do this by putting the word *static* in front of the variable declaration. For example, if the declaration "static int count;" is located in a function, count will be visible only in the function but will hold its value of "live" all the time that the program is running. If you put the word static in front of a variable declaration that is outside of main, the effect is to make the variable accessible or visible only in the source file where it is declared.

Another useful storage class for variables is *register*. You might, for example, declare a variable in a function with a statement such as "register int index;." The term register at the start of this declaration asks the compiler to assign this variable to one of the 8086 registers. The reason for doing this is that it is much faster to, for example, increment the contents of a register than it is to increment the contents of a memory location dynamically allocated to an automatic variable. If all registers are in use, the compiler will ignore the register storage request and treat the variable as a normal automatic variable.

Functions also have storage classes. By default, functions are extern or global. This means that they can be accessed from other files. To access a function from another file, you write a copy of the function prototype in that file and put the word extern in front of it.

If you give a function the storage class static, the function is accessible only from within the file where it is defined.

To summarize the different storage classes and their characteristics, Figure 12-22 shows examples of each. You can use these examples to help you decide which storage class to use for particular applications in your programs.

FUNCTIONS AND ARRAYS

One of the main reasons to learn about C pointers is so that you can use them with functions. As we said before, if you want a function to modify the value of a variable, you must pass the function a pointer to the variable. In Figure 12-21*b* we showed you how to pass a simple variable pointer and in Figure 12-12 we showed you how to pass an array pointer to the predefined scanf function. Now we need to show you how to pass array pointers to functions you write.

Figure 12-23 shows how you can declare, define, and call a function to add profit to costs instead of doing the operations in main as we did in previous examples. The first section of main prompts the user and then calls scanf to read in 10 costs and put the 10 values in a float array called cost. Remember that scanf requires a format specifier and a pointer to where you want to put the value read. In Figure 12-12 we used a declared pointer as the argument for scanf, but here we use the expression (cost + *i*) as a pointer to the desired element in cost. Cost is a pointer to the first element in the array, and as we explained earlier, (cost + *i*) is a pointer to element *i* in the array. When the compiler performs pointer arithmetic on the expression (cost + *i*), it automatically multiplies *i* times the number of bytes in the data type so that the computed pointer accesses the desired element.

After all the values are read into the cost array, we call the function add_profit to compute the selling price for each and print the results. The add_profit function is type void, because it does not return a value directly to main. The expression in the parentheses of the add_profit function header declares a float pointer called

```
VARIABLE EXAMPLE                LIFETIME      ACCESSIBILITY
int tempf;                      program       all source files
static int tempc;               program       this file only
extern int book_total           program       defined in another file
int c2f(inc c);                 program       all source files
static int f2ç(intf)            program       this source file only

void main ()
{
int count;                      block         block and sub blocks
                                              after declared
static int interrupt_cnt; program             block and sub blocks
                                              after declared
register int index              block         block and sub blocks
                                              after declared
```

FIGURE 12-22 Examples of variable and function storage classes.

```
/* C PROGRAM F12-23.C */

/* Passing array pointers to functions */

float cost[10], prices[10];              /* array declarations */
                                         /* function declaration or prototype */

void  add_profit(float *pp, float *cp, int count);

void main ()
{
     int i;
      int number=10;
      printf("Enter %d costs. After each cost press enter.\n",number);

      for(i=0; i < number; i++)          /* read in costs */
        scanf("%f", (cost+i));

      add_profit(prices, cost, number); /* function call */
}  /* end of main */


/* function definition */

void add_profit(float *pp, float *cp, int count)
      {
      int i;
      for(i=0; i < count; i++)
        {
        *(pp+i) = *(cp+i) + .25 * *(cp+i);
        printf("cost=%6.2f, price=%6.2f \n",*(cp+i),*(pp+i));
        }
      }
```

FIGURE 12-23 Program showing how to pass array pointers to functions.

pp that will be used to receive a pointer to prices and a float pointer called cp that will be used to receive a pointer to cost. The header also declares an int which will receive the number of elements in the array. Here's why we declared these three.

The procedure reads a value from the array cost, computes the selling price, and puts the result in the array prices. Since we are changing values in the prices array, we have to pass the function a pointer to prices. For this simple example, however, we are not modifying the values in cost, so we did not actually have to pass a pointer to cost. The array cost could have been accessed directly from the function. (Remember, cost is declared outside of main, so it is extern and accessible globally.)

If you refer to cost directly in the function, then the function will work only with values from the array cost. We passed both the source and the destination pointers to the function so that the function will work with any array of costs and any array of prices. Likewise, we pass the number of elements in the array to the function. The for loop in the function uses this passed number instead of a fixed number to determine how many elements to process. The function then can process arrays with any number of elements up to the limit of int, which is +32,767.

In a more realistic program you might declare the arrays large enough to hold 1000 or more elements and then get the value for number by counting how many costs a user actually entered before entering an EOF character (Ctrl Z). The main point we are trying to make here is that by passing pointers and lengths to functions instead of passing directly named variables, you make the function more universally useful or "portable."

Since the name prices is a pointer to the prices array and the name cost is a pointer to the cost array, the actual call of add_profit in Figure 12-23 passes prices to pp, cost to cp, and number to count.

The example we have just discussed shows you how to write a function which accesses two one-dimensional arrays. Figure 12-24, page 422, shows how you can declare, define, and call a function which accesses the elements in a two-dimensional array. Specifically, the function in this program converts each Celsius temperature in a two-dimensional array of temperatures to its Fahrenheit value. This program is simply an extension of the program in Figure 12-20*a*.

In Figure 12-24 the first for loop in main reads the max and min Celsius temperatures for 7 days and puts them in the first two columns of a 7 × 3 array. The second for loop in main computes the average temperature for each day and writes the result in the third column of the appropriate row in the array.

Once all the Celsius values are in place, we call the function c2f to convert each Celsius value to its

```
/* C PROGRAM F12-24.C */
/* Program to read max and min Celsius temperatures, compute average,
   convert all values to Fahrenheit, and display results */

#include <stdio.h>
int ctemps[7][3];
int ftemps[7][3];
void c2f(int ct[][3], int ft[][3], int rows);     /* function declaration */

void main()
{
     int days = 7;
     int i, j;                          /* note i and j separate variables in main and c2f */
     for (i=0; i<days; i++)
         {
          printf("Enter max Celsius temp for day %d,"
                "then min Celsius temp for day %d.\n", i+1,i+1);
          for(j=0; j<2; j++)                                   /* read max, then min */
          scanf ("%d", &ctemps[i][j]);
         }

     for(i=0; i<days; i++)
     {
     ctemps[i][2] = (ctemps[i][0] + ctemps[i][1])/2; /* average */
     printf("Celsius temperatures for day %d: max = %d min = %d "
        "av = %d \n", (i+1), ctemps[i][0], ctemps[i][1], ctemps[i][2]);
     }
     c2f(ctemps,ftemps, days);          /* call c2f function */
     for(i=0; i<days; i++)
     printf("Fahrenheit temperatures for day %d: max = %d min = %d "
          "av = %d \n", i+1,
          ftemps[i][0], ftemps[i][1], ftemps[i][2]);
} /* end of main */

   /* define c2f function */
void c2f(int ct[][3], int ft[][3], int rows)
     {
     int i, j;                           /* note these variables different from I,J in main */
     for(i=0; i < rows; i++)
          for(j=0; j<3; j++)
          ft[i][j] = 9*ct[i][j]/5 + 32;
     }
```

FIGURE 12-24 Program using pointers and functions with a two-dimensional array.

Fahrenheit equivalent and put the results in an array called ftemps. As with the previous example, we want to pass pointers to the two arrays and pass the length of the arrays so that the function is as versatile as possible.

The expression int ct[ ][3] in the c2f function header in Figure 12-24 shows one way to declare the pointer needed to receive a pointer to a two-dimensional array. The empty brackets between ct and [3] indicate that ct is a pointer to an array of three elements. When we call the c2f function, we pass ctemps as the actual argument. As shown in Figure 12-20c, the name ctemps is a pointer to the first three-element array, temps[0], so the call gives the c2f access to the first three-element array. In the c2f function a nested for loop is used to access the elements in ctemps[0], ctemps[1], ctemps[2], etc.

In the same way the int ft[ ][3] expression in the c2f function header declares another pointer to an array of three elements. This formal parameter is used to receive a pointer to ftemps during the call. Incidentally, you can declare a pointer to a three-dimensional array with an expression such as float hrs_worked[ ][12][31]. The trick here is to simply leave the first set of brackets after the array name empty.

Another method of declaring the formal argument for passing the ctemps pointer to the function is with the expression int (*ct)[3]. This expression likewise declares ct as a pointer to a three-element array. The parentheses around *ct are required to indicate that you are declaring a pointer to an array. The expression int *ct[3] declares an array of three pointers which each point to int-type variables.

To summarize the operation of all this, the c2f(ctemps, ftemps, days) statement in Figure 12-24 calls the function. The ctemps in the function call passes a pointer to the ctemps array to the function pointer variable ct. The ftemps in the function call passes a pointer to the ftemps array to the function pointer variable ft. The days in the function call passes the value of the variable days to the function variable called rows. The function uses these passed values and a nested for loop to read an element from ctemps, compute the Fahrenheit equivalent, and write the result to the same element in ftemps. Note that since the number of rows is a variable in the function, the function can be called to process any number of three-element arrays.

DECLARING AND USING POINTERS TO FUNCTIONS

In the preceding sections we have shown you how to declare pointers to simple variables and pointers to arrays. You can also declare and initialize a pointer to a function. This is an advanced technique and it is unlikely that you will use pointers to functions in your initial programs. However, we want to show you a couple of examples so that you will recognize them in someone else's programs. Here is how you could declare a pointer to the c2f function in Figure 12-21*c* and call the function using the pointer instead of using a direct call.

```
int c2f (int c);          /* declare the function c2f */
int (*convert) (int c);   /* declare a pointer to a function */
convert = c2f;            /* initialize the pointer to point
                             to c2f */
tempf = (*convert)        /* call c2f with pointer and pass */
   (tempc);               /* value of temp c to the function */
int c2f(int c)            /* c2f function definition header */
```

The basic function declaration and definition here are the same as those in Figure 12-21*b*. The second statement declares a pointer called convert that points to a function. The key to recognizing that convert is a pointer to a function is the double set of parentheses in the declaration. The int at the start of the declaration indicates that the function pointed to returns an int value. The int c in the second set of parentheses indicates that the function pointed to expects to receive an int value. The parentheses around the name of the function pointer are required to indicate that convert is a pointer to a function. The statement int *convert(int c);, which does not have these parentheses, declares a function that returns a pointer to an int value.

The tempf = (*convert)(tempc); statement calls the c2f function using the pointer called convert. The term *convert represents the contents of convert, which we initialized with the address of the c2f function. The value of tempc is passed to the function, and the int value returned by c2f is assigned to tempf.

Now that you know much more about functions, in the next section we will take a closer look at some of the predefined functions available to you in libraries.

C Library Functions

INTRODUCTION

Throughout this chapter we have used predefined functions such as printf( ), scanf( ), and getch( ) in many of the example programs. The functions we have used are just a small sample of those available. Turbo C++ comes with a *Run Time Library* containing over 450 predefined functions and macros. These library functions allow you to perform I/O operations with a variety of devices, dynamically allocate memory in a program, produce graphics displays, read from and write to disk files, perform complex mathematical computations, etc. For many applications you can use one of these predefined functions instead of writing your own function. The source code for all these functions is available from Borland, so if the predefined function does not quite fit your application, you can modify a copy of the source code for the function to produce a custom version which does.

The declarations or prototypes for the predefined functions are contained in files called header files or include files. These files have names such as stdio.h, string.h, math.h, graphics.h, and alloc.h. The preprocessor #include directive tells the compiler which header files to search for the predefined functions you use in a program. The directive #include<stdio.h>, for example, tells the compiler to look in the header file called stdio.h to find the prototypes for functions such as printf( ), scanf( ), and getch( ).

The actual codes for the predefined functions are contained in library (.lib) files. When you call a function, the object code for the function gets linked with the code for the rest of your program when the .exe file is created.

In the following sections of the chapter we review the functions we have used previously and show some more functions and examples that you may find useful in your programs. In later chapters we show you how to use other predefined functions for graphics, disk file, and communications programs. To help you refer to the examples here, we have separated them according to the type of operation they perform. For discussions of all 450+ functions and macros consult the Turbo C++ Reference Guide.

KEYBOARD INPUT FUNCTIONS

Function	*Prototypes in stdio.h*	
getch( )	int getch(void)	/* read char as soon as pressed */
getche( )	int getche(void)	/* read char and echo to CRT */
getchar( )	int getchar(void)	/* wait for Enter, read char */
gets( )	char *gets(char *s)	/* reads characters from keyboard until Enter and writes string to location pointed to by s. Reads spaces and tabs.*/
scanf( )	int scanf(const char *format,[address, . . .])	

Scanf reads characters from the keyboard until it reads a blank, a tab, or an Enter. Data read in is formatted according to the format specifier in the call and written to the address passed in the call. The three dots after address indicate that the number of arguments to be passed to scanf is variable. This means that you can include several format specifiers and several addresses in one scanf call to read in multiple values.

Scanf normally returns the number of values read and stored. If the first entered character that scanf reads cannot be converted to the specified format, scanf will not store the value, and it will return a value of 0. For example, if the scanf call statement contains a %f format specifier and you accidentally enter a T, scanf will terminate and return a 0.

The input loop in Figure 12-12 can be rewritten as follows to make sure that the pointer does not get incremented if no value was written to one of the elements in the array:

```
for(i=0;i<10; i++)
{
      if((scanf("%f",cpntr)) = =0)
      {
      i--;                /* correct index value */
      fflush(stdin);      /* clear unread characters from
                             keyboard buffer */
      continue;           /* skip rest of loop actions */
      }
      else cpntr++;
}
```

The continue statement here will cause the cpntr++ to be skipped over in this trip through the loop if the value returned by scanf is zero.

NOTE: This cure does not work if an illegal character is entered in any but the first digit position.

OUTPUT FUNCTIONS

Function *Prototype in stdio.h*

putchar( ) int putchar (int c) /* outputs passed character to screen. Returns −1 (EOF) if error. */

puts( ) int puts(const char *s)

Puts sends a null terminated string pointed to by s to the screen. If an error occurs, puts returns a value of −1 (EOF). For outputting simple strings, puts uses much less memory and time than printf.

printf( ) int printf(const char *format, [argument, . . .]);

As shown by the many examples in the preceding programs, the format here consists of text and format specifiers. The arguments are a list of variables, one for each format specifier. The general form of the format specifiers is as follows:

% flags width . precision [F,N, h, l, L] type

flags = output justification, numeric signs and other
− = left justify printed digits
+ = print + or minus sign in front of value
blank = positive values start with blank instead of +
width = total number of digits left of decimal point
[F, N, h, l, L] = override default size of argument with
F = far pointer, N = near pointer, h = short int,
l = long, L = long double
type = conversion specifier as shown in Figure 12-11

Consult the Turbo C++ Reference Manual for a complete explanation of the print controls in printf.

fprintf( ) int fprintf (FILE *stream, constant char *format [,argument, . . .])

With the proper setup fprintf( ) will send program output to the printer instead of to the CRT screen. As we discuss further in a later chapter, we often think of data going to or coming from a disk file as a "stream." The same term can be used to refer to data going to the CRT. The fprintf( ) function allows a stream of data to be sent to the printer. Figure 12-25 shows how this function can be used to send the output of our old prices program to the printer.

Before you can call the fprintf( ) function, you must use the predefined setmode( ) function to tell the compiler that you are going to send a text file to the printer. The 0004 in this call is a "handle" which identifies the printer, and the O_TEXT is a predefined term for text mode. The prototype for setmode( ) is in fcntl.h, so we put #include<fcntl.h> at the top of the program.

The fprintf( ) function call is the same as a call to printf, except that we include the term stdprn before the usual printf arguments. The term stdprn tells fprintf to direct the data stream to the standard printer device. Incidentally, the \f in the final fprintf statement is a formfeed character, which tells the printer to advance to the top of the next page.

STRING FUNCTIONS

Function *Prototype in string.h*

strcat( ) char *strcat(char *dest, cons char *src)

Strcat( ) adds a copy of string pointed to by src to the string pointed to by dest. and returns a pointer to the start of the combined string.

strchr( ) char *strchr(const char *s, int c);

Strchr scans a string pointed to by s for the first occurrence of c. Strchr returns a pointer to the first occurrence of c or returns a null if c was not found in the string.

strlen( ) size_t strlen(const char *s);

Strlen returns length of string pointed to by s.

strcmp( ) int strcmp(const char *s1, const char *s2);

Strcmp compares each character in s1 with the corresponding character in s2. Strcmp( ) returns 0 if the two strings are equal, a positive number if s1 is greater than s2, and a negative number if s2 is greater than s1. The

```
/* C PROGRAM F12-25.C   */
/* Sending program output to a printer */

#include <stdio.h>
#include <fcntl.h>

int cost[] = { 20,28,15,26,19,27,16,29,39,42 };     /* array of 10 costs */
int prices[10];                                      /* array to hold 10 prices */

void main()
{
     int index;
     setmode(0004, O_TEXT);
     for (index=0; index <10; index++)               /* for loop to compute */
     prices[index] = cost[index] + 15;               /* 10 prices */

     for (index=0; index <10; index++)               /* for loop to display results */
       fprintf(stdprn,"cost = %d, price = %d, \n",
                      cost[index], prices[index]);
     fprintf(stdprn,"\f");
}
```

FIGURE 12-25 Program using predefined fprintf function to send program output to a printer instead of to the CRT.

stricmp( ) function is the same as strcmp( ), except that it ignores the case of the characters in the strings. Figure 12-26 shows how you can use the stricmp( ) function to implement an improved version of the password check program from Figure 5-3.

At the start of the program we declare the required character arrays and a counter. Then we prompt the user and use gets( ) to read the response. The while loop compares the value returned by stricmp with 0 to see if the entered password is correct. If the password is correct, execution exits the while loop and goes on to the if structure. If the entered password is incorrect, the while loop gives the user two more tries to enter the correct password before going on to the if structure.

If the value returned by stricmp( ) is equal to zero, execution will simply fall through the if structure and print the welcome message. If the user did not get the password correct in three tries, then the if structure prints a message, sounds an alarm, and exits. In a more realistic program you would probably call a function which locks up the machine at this point instead of doing a simple exit.

```
/* C PROGRAM F12-26.C    */
/* Password program in C */
#include<stdio.h>
#include<string.h>
void main()
{
 char password[] = "failsafe";
 char input_word[8];
 int try = 0;
 printf("Please enter your password.\n");
 gets(input_word);
 while(stricmp(password,input_word) != 0 && try++ <2)
  {
  printf("Entered password is incorrect,try again.\n");
  gets(input_word);
  }
if(stricmp(password,input_word) != 0)
  {
  printf("This computer does not know you!");
  /* alarm() *//* call ASM function to sound alarm */
  exit();
  }
  printf("Welcome, what can I do for you?");
}
```

FIGURE 12-26 Program using predefined string function to compare passwords.

MATH FUNCTIONS

Function	*Prototype in math.h*
sqrt( )	double sqrt (double x);

Sqrt( ) returns the positive square root of x. If x is negative, sqrt returns zero. We don't have space here to discuss the prototypes for the many 8087 type math functions found in math.h. However, to keep a promise we made earlier, Figure 12-27, page 426, shows how the Pythagoras program from Chapter 11 can be written in C.

Remember, this program calculates the value of the hypotenuse of a right triangle by taking the square root of the sum of the squares of the two legs. In the program in Figure 12-27 we call the predefined function sqrt( ) to take the square root. We pass sqrt a value which is equal to side_a squared + side_b squared. Sqrt returns the square root of the sum and assigns it to side_c. Note that we wrote a #include<math.h> directive at the start of the program to tell the compiler where to look for the prototype of the sqrt( ) function.

When the compiler compiles this program, it will use the default mode of "emulator" for the instructions

```
/* C PROGRAM F12-27.C  */
/*PYTHAGORAS REVISITED */

#include <stdio.h>
#include <math.h>
void main (  )
{
   float side_a, side_b, side_c;
   side_a = 3.0;
   side_b = 4.0;

   side_c = sqrt (side_a * side_a + side_b * side_b);
   printf("side a = %2.2f side b = %2.2f
           side C = %2.2f\n", side_a, side_b, side_c);
}
```

FIGURE 12-27 C version of 8086/8087 Pythagoras program in Figure 11-22.

which act on floating-point numbers. When you run the program, a predefined function determines if your system contains an 8087 or 80287. If an 8087 is present, the program will use 8087 instructions to implement floating-point operations in the program. If your system does not contain an 8087, the program will use floating-point library functions which emulate the 8087 instructions. If you are sure that a floating-point program will be run only on systems which have 8087s present, you can work your way through the menu path Options->Compiler->Code generation in the Turbo C++ IDE and toggle the Floating-point line to 8087/80287. This will shorten the length of the .exe program produced, because the emulation functions do not have to be included.

Writing Programs Which Contain C and Assembly Language

INTRODUCTION

The C language is very useful for writing user-interface programs, but code produced by a C compiler does not execute fast enough for applications such as drawing a complex graphics display on a CRT. Therefore, system programs are often written with a combination of C and assembly language functions. The main user interface may be written in C and specialized, high-speed functions written in assembly language. These assembly language functions are simply called from the C program as needed.

Also, when writing a program that is mostly assembly language, you may find it useful to call one of the predefined C functions to do some task that you don't want to take the time to implement in assembly language.

The main points you have to consider when interfacing C with assembly language are

1. How do you call a desired function?
2. How do you pass parameters to the called function?
3. How are parameters passed back to the calling program from the function?
4. How do you declare code and data segments in the function so that they are compatible with those in the calling program?

The easiest way to answer these questions is to look closely at how the compiler does each one. Here's how you get a look at how a compiler treats a C program.

Earlier in the chapter we described how you can use the Turbo C++ IDE to compile, run, and debug programs. In addition to the compiler in the IDE, the Turbo C++ toolset has a separate compiler called tcc. The tcc compiler has a few advanced features that the integrated compiler doesn't have. The tcc compiler, for example, will compile a C program to its assembly language equivalent. The command tcc -S 12-28a.c, for example, will produce a file called 12-28a.asm which contains the assembly language equivalent for the specified C source file. The C source program statements are included as comments in the .asm file.

THE ASSEMBLY LANGUAGE EQUIVALENT OF A C PROGRAM

Figure 12-28*a* shows a simplified version of the temperature conversion program in Figure 12-21*b*, and Figure 12-28*b* shows an edited version of the .asm program produced from it by tcc. To make the program easier to follow, we removed all the debug information normally put in by the compiler, shortened the list of DBs at the end of the program, and added some comments. Read the C program in Figure 12-28*a*, skim through the .asm version in Figure 12-28*b* to see how much you can intuitively understand, and then come back to the discussion here to get more details. The analysis of this

```
/* C PROGRAM F12-28A.C /*
/* Simple temperature conversion function
   definition and call */

#include<stdio.h>
int tempc = 25, tempf;/* external (global) variables*/

int c2f(int c);         /* declare function c2f which
                           returns an int value */
void main()
{
 tempf = c2f(tempc);   /* call c2f function, pass value
                          of tempc to function. Returned
                          value assigned to tempf */

 printf("Celsius = %d,Fahrenheit=%d \n", tempc,tempf);
}  /* end of main */

int c2f(int c)          /* define function c2f.
                           Note no ; at end */
     {
     int f;             /* automatic (local) variable */
     f = 9*c/5 + 32;
     return (f);
     }
```

(*a*)

FIGURE 12-28 (*a*) Simplified version of Figure 12-21*b*.

```
;8086 PROGRAM F12-28B.ASM
_TEXT   SEGMENT     BYTE PUBLIC 'CODE'
DGROUP  GROUP       _DATA,_BSS          ; Assign same start to segments
        ASSUME      CS:_TEXT, DS:DGROUP, SS:DGROUP
_TEXT   ENDS

_DATA   SEGMENT WORD PUBLIC 'DATA' ; Initialized variables here
_TEMPC  LABEL   WORD               ; Declare and init TEMPC
        DW      25
_DATA   ENDS

_TEXT   SEGMENT BYTE PUBLIC 'CODE' ; Code always in _TEXT segment
_MAIN   PROC      NEAR
    PUSH  WORD PTR DGROUP:_TEMPC     ; Pass value of TEMPC on stack
    CALL  NEAR PTR _C2F              ; Call C2F function
    POP   CX                         ; Increment SP over TEMPC arg
    MOV   WORD PTR DGROUP:_TEMPF,AX  ; Save TEMPF returned in AX
    PUSH  WORD PTR DGROUP:_TEMPF     ; Put value of TEMPF on stack
    PUSH  WORD PTR DGROUP:_TEMPC     ; Put value of TEMPC on stack
    MOV   AX,OFFSET DGROUP:Sa        ; Put pointer to text string
    PUSH  AX                         ;     on stack
    CALL  NEAR PTR _PRINTF           ; Call PRINTF fucntion
    ADD   SP,6                       ; Increment SP over three passed arguments
    RET                              ; Return from main
_MAIN     ENDP

_C2F      PROC      NEAR             ; C2F function definition
    PUSH  BP                         ; Save old BP
    MOV   BP,SP                      ; Copy of SP To BP
    PUSH  SI                         ; Save SI reg because used here
    MOV   AX,WORD PTR [BP+4]         ; Get value of TEMPC from stack
    MOV   DX,9                       ; prepare to multiply by 9
    MUL   DX                         ; Multiply value of TEMPC by 9
    MOV   BX,5                       ; Prepare to divide result by 5
    CWD
    IDIV  BX                         ; Do division, int result in AX
    MOV   SI,AX                      ; SI used for local variable F
    ADD   SI,32                      ; Add 32 to F
    MOV   AX,SI                      ; Value of F returned in AX
    POP   SI                         ; Restore SI
    POP   BP                         ; Restore old BP value
    RET                              ; Return to main
_C2F      ENDP
_TEXT     ENDS

_BSS      SEGMENT WORD PUBLIC 'BSS'  ; Uninitialized variables here
_TEMPF    LABEL   WORD               ; Declare TEMPF
          DB      2 DUP (?)
_BSS      ENDS
```

(*b*)

FIGURE 12-28 (*Continued*) (*b*) Assembly language equivalent of C program in 12-28*a* produced by tcc compiler with -S switch. (*Continued on next page.*)

```
_DATA     SEGMENT WORD PUBLIC 'DATA'    ; Text string and format
S@        LABEL     BYTE                ; Specifiers for PRINTF here
          DB      67
          DB      101
           :                            ; List shortened to save space
          DB      116
          DB      32
          DB      61
          DB      100
          DB      32
          DB      10
          DB      0
_DATA     ENDS

_TEXT     SEGMENT     BYTE PUBLIC 'CODE'
          EXTRN       _PRINTF:NEAR      ; Let compiler know PRINTF()
_TEXT     ENDS                          ; function is external

     PUBLIC       _TEMPF                ; Make extern variables and
     PUBLIC       _MAIN                 ; functions public
     PUBLIC       _TEMPC
     PUBLIC       _C2F
     END
```

(*b*)

FIGURE 12-28 (*Continued*) (*b*) Assembly language equivalent of C program in 12-28*a* produced by tcc compiler with -S switch.

program should help you better understand some of the earlier discussions of passing arguments to functions and variable storage classes.

The C program in Figure 12-28*a* calls our c2f function to compute the Fahrenheit equivalent of 25°C and calls the predefined printf function to display the result. The first feature we need to talk about in the assembly equivalent for this program is how the segments are defined and grouped.

Turbo C allows you to compile a program for any of six *memory models*. The six memory models are tiny, small, medium, compact, large, and huge. The memory model used determines the location of segments in memory and the size pointers used to refer to code and data. Here is a short discussion of each,

Tiny—All four segment registers are set to the same physical address, so only 64 Kbytes are available for all code and data. Since everything is in one 64-Kbyte space, near pointers are used for all code and data references. The tiny model is used to generate .com-type programs which automatically get loaded into memory at 100H.

Small—This model uses one 64-Kbyte code segment. One 64-Kbyte segment is shared by the data segment, the stack segment, and the extra segment, so these segments all start at the same address. This memory model is the default for the Turbo C + + compilers. Near pointers are used for all code and data references.

Medium—Far pointers are used for references in code, so code references can be anywhere in the 1-Mbyte address space. The data segment, extra segment, and stack segment share one 64-Kbyte space, so near pointers are used for data references.

Compact—This model uses one 64-Kbyte code segment, so near pointers are generated for code references. Far pointers are generated for data references, so data can be accessed anywhere in the 1-Mbyte range.

Large—Far pointers are used for both code and data references, so both have a 1-Mbyte range. If a program has a code file or a data file larger than 64 Kbytes, however, the file must be broken into files smaller than 64 Kbytes and the resulting files linked.

Huge—Huge is similar to the large model, except that the far pointers are always normalized. A far pointer is normalized by generating the 20-bit physical address from the segment and offset and then using the upper 4 nibbles of the physical address as the segment and the lower nibble of the physical address as the offset. A pointer reference of 4057:3244 produces a 20-bit address of 437B4 and a normalized pointer value of 437B:0004. The advantage of normalized pointers is that they can be accurately compared in expressions using the = =, ! =, =, > =, <, and < = operators.

We used the default memory model to compile the program in Figure 12-28*a*, so the data, stack, and extra segments all share one 64-Kbyte address space. The DGROUP GROUP _DATA, _BSS statement at the top of Figure 12-28*b* groups the logical segments _DATA and _BSS together in a group called DGROUP. The ASSUME statement just after this indicates that DS will be initialized to point to DGROUP and SS will also be initialized to point to DGROUP.

Note that the assembly language for this program does not show any instructions for initializing the segment registers and the stack pointer register. These instructions are contained in a special startup section of code that is linked with your program when the .exe file is created.

If you look again at Figure 12-28*b*, you can see that the _TEXT segment is used to hold program instructions. The _DATA segment is used to hold initialized extern variables and text strings such as those used in printf statements. The _BSS segment is used to hold uninitialized extern variables. You should use these same conventions when you write an assembly language function to be called from a C program.

The next point to consider here is how C passes arguments to a function. If you call an .asm function from a C program, this is the way the arguments will be passed to the function. If you call a C function from an .asm program, this is how you have to pass arguments to the C function.

C passes almost all arguments to functions by pushing them on the stack. The first instruction in main in Figure 12-28*b* pushes the value of TEMPC on the stack to pass to C2F. As we said earlier, this call just passes a copy of CTEMP to the function, so the function cannot change the actual value of CTEMP. Remember, if you want a function to change the value of a variable, you pass the offset of the variable to the function.

Now let's look at how the function accesses the CTEMPS value passed to it on the stack. The process here is the same one we introduced you to in Figure 5-17. We first save the old value of BP by pushing it on the stack and then copy the value of SP to BP so that BP is a second pointer to the stack. As we told you in Chapter 5, the easiest way to keep track of where everything is in the stack is with a simple stack map such as that in Figure 12-29. We pushed ctemps on the stack in main, the return address (IP) got pushed on the stack during the call, and BP got pushed on the stack at the start of the C2F function. The BP register then points to the stack at the location where the old value of BP is stored. You can access any value on the stack by simply adding a displacement to BP. The value of CTEMPS is in the stack at [BP+4], so the instruction MOV AX,WORD PTR[BP+4] will copy the value of CTEMPS to AX. The 8086 arithmetic instructions after this perform the specified computations.

Note that the compiler assigned the local variable named f in the C program to the SI register, even though we did not tell it to make f a register variable. If you declare more than two automatic or register variables, the compiler will allocate space for them on the stack below BP, as shown in Figure 12-29. Since they are dynamically allocated on the stack, automatic variables are re-created each time a function is called.

```
                              Stack map

stack pointer           stack

before push      →

after push       →     ctemps    ← [BP+4]

after call       →       IP      ← [BP+2]

after push BP    →       BP      ← [BP]

after push SI    →       SI      ← [BP-2]

automatic        →       __      ← [BP-4]
variable

automatic        →       __      ← [BP-6]
variable
```

FIGURE 12-29 Stack map showing use of BP to access arguments passed to a function on the stack.

The computed value of f is returned to the calling program in the AX register. If the function were returning a 32-bit value, it would return the high word in DX and the low word in AX.

If you look at the stack map in Figure 12-29 again, you should see that when execution returns to main from the procedure, SP will be pointing to the value of ctemps in the stack. The POP CX instruction in main will "clean up" the stack by incrementing SP to its initial value. Since C doesn't use the CX register for anything special, we don't care about the value put in CX by the POP CX instruction.

The next part of the program pushes the specified arguments on the stack and then calls printf to display the results of the computation. The C compiler pushes arguments on the stack in the reverse order from the order they are written in the function call parentheses. For our example here, then, the value of ftemp will be pushed on the stack first. Next the value of ctemp will be pushed on the stack. Finally, a pointer to the text string in the printf( ) call will be pushed on the stack. This final step is done with the two instructions MOV AX,OFFSET DGROUP:S@ and PUSH AX.

The printf( ) function is obviously not present in this source module. The statement EXTRN _PRINTF: near the bottom of Figure 12-28*b* indicates that the code for this function will be linked later. When execution returns from the printf function, the ADD SP,6 instruction cleans up the stack by incrementing it up over the three arguments passed to printf on the stack.

Finally in Figure 12-28*b*, note that the variables and functions declared outside of main are made public so that they can be accessed from other source modules. As we showed you in Chapter 5, if you want to access one of these from another source module, you have to declare it extrn in that module.

Now that you have some ideas about how a C compiler "thinks," let's talk about how you can use this to write assembly language functions you can call from your C programs and how you can call C functions from your assembly language programs.

A PROGRAM WITH C AND ASSEMBLY LANGUAGE MODULES

Figure 12-28*b* shows you almost everything you need to know to interface C and assembly language, but to make it a little clearer, Figure 12-30 shows a C program which calls two assembly language functions and also shows the two assembly language functions. To show a C function call from assembly language, one of the assembly language functions calls the predefined C function, printf( ).

In the C program in Figure 12-30*a*, we put the term extern in the two function declarations to let the compiler know that these functions are in another source module. We then call the functions by name and pass any required arguments, just as we would call C functions.

In the assembly language part of the program in Figure 12-30*b*, we declare segments using the names shown in Figure 12-28*b*. Note that you put underscores (_) in front of all segment names, function names, and variable names in the assembly language module. This is required for compatibility with the C compiler conventions.

The c2f function in Figure 12-30*b* is exactly the same as that produced by the compiler in Figure 12-28*b*. It is very common practice to write a function in C, compile the function to its assembly language equivalent, and then "hand optimize" the .asm equivalent for maximum efficiency in the specific application. As we will show you, the .asm file can be assembled and the resulting object file linked with the object file for the mainline program.

The show function in Figure 12-30*b* calls printf to display the Celsius temperature, the Fahrenheit temperature, and appropriate text. We declare the text in the data segment with a simple DB statement. The 0AH at the end of the declaration represents a carriage return, and the 00H is a NULL character required as a terminator on the string. From the string you can see that we need to pass three arguments to printf, just as we did in Figure 12-28*a*. The three arguments are a pointer to the string, the value of tempc, and the value of tempf. The three push statements in Figure 12-30*b* put these arguments on the stack in reverse order as required by the C calling convention. When execution returns from printf, we add six to SP to increment it up over the three arguments we passed to printf.

A very important point to observe in Figure 12-30 is the use of the extern or extrn directives and the use of the public directive. In the C program we use the extern directive to tell the compiler that c2f and show are in another source module. In the assembly module in Figure 12-30*b* we use the public directive to make the procedures c2f and show accessible to other source modules. Note that the public declarations are put in the code segment. Also in Figure 12-30*b* use the extrn directive to tell the assembler that the variables tempf and tempc are defined in another source module. Note that the extrn directives are put in the segments where the named variables are found.

As we told you in Chapter 5, the rules here are very simple. You declare a function or variable public in the module where it is defined if you want other modules to be able to access it. You use the extern or extrn directive to tell the assembler/compiler that a function or variable is located in some other source module.

SIMPLIFIED SEGMENT DIRECTIVES

Newer versions of TASM and MASM allow you to use a simplified set of segment directives in assembly language programs. You can use these simplified segment directives in many stand-alone assembly language programs, but their main use is in writing assembly language modules which interface with high-level language program modules.

Figure 12-30c shows in skeleton form how the assembly language module from Figure 12-30*b* can be written using these simplified directives. The DOSSEG directive at the start tells the linker to put the segments in an order which is compatible with DOS and high-level languages. Basically the order is code segment, data segment containing initialized variables, data segment containing uninitialized variables, and stack segment.

The .MODEL directive tells the assembler to use the SMALL memory model, which consists of one 64-Kbyte code segment and one 64-Kbyte data segment. This is the default model for the Turbo C++ compiler.

The .CODE directive sets up the code segment. With this directive the assembler automatically gives the code segment the name required by the memory model used and generates the required ASSUME directive. For the small memory model the code segment will be assigned the name _TEXT, as shown in Figure 12-30*b*.

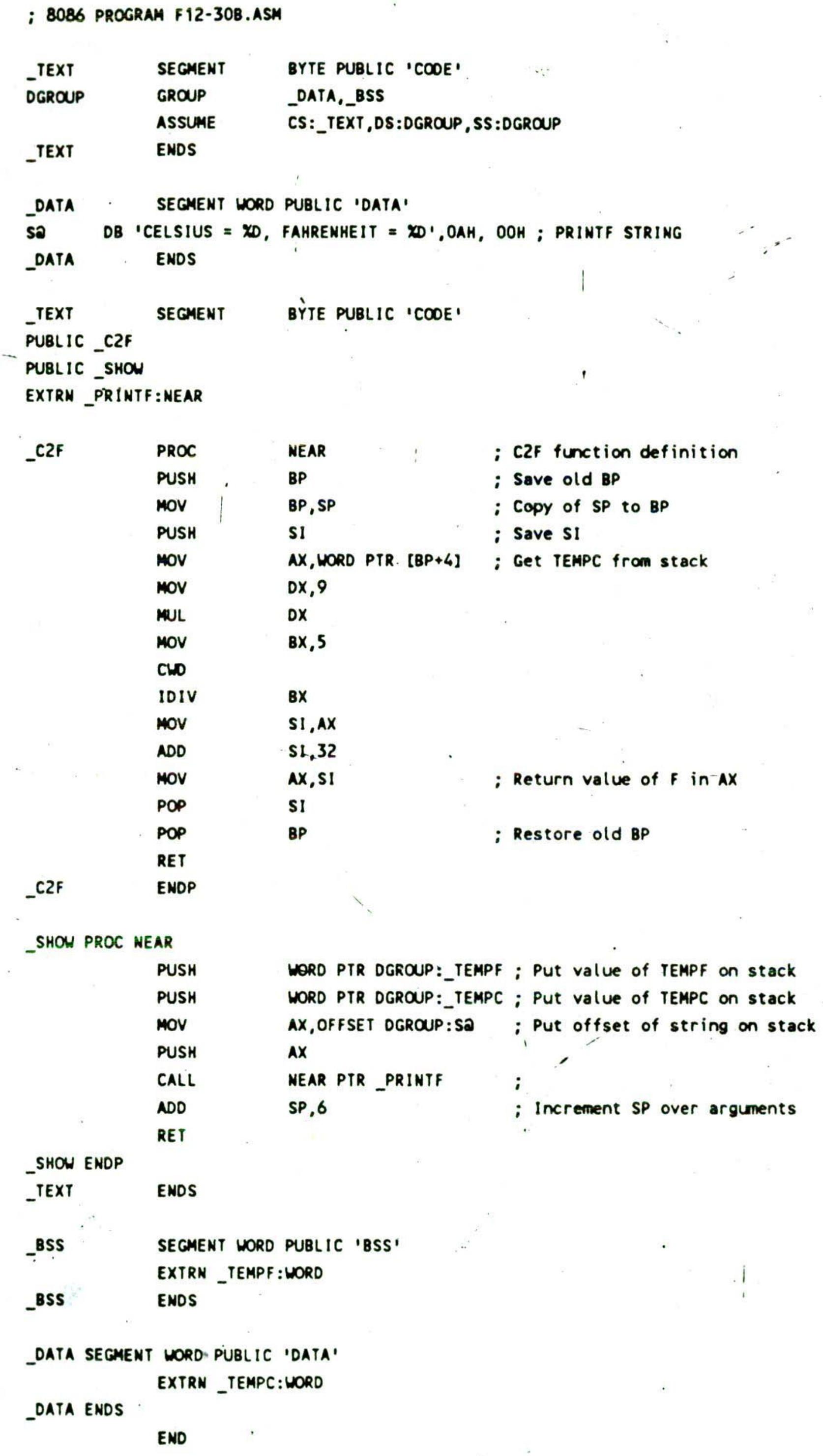
In a similar way the .DATA directive declares a segment for initialized variables and the .DATA? directive declares a segment for uninitialized variables. In a small model program the assembler will automatically "group" these two segments, as we described for the standard segment directive version in Figure 12-30*b*. Incidentally, you do not need to declare a stack or initialize the stack

```
/* C PROGRAM F12-30A.C  */
/* Temperature conversion function */

#include<stdio.h>
int tempc = 25, tempf; /* external (global) variables*/
int extern c2f(int c); /* declare function c2f which */
                       /* returns an int value */
void extern show(void);/* function show is in
                          another module */
void main()
{
 tempf = c2f(tempc);  /* call c2f function, pass
                         value of tempc to function.
                         Returned value assigned
                         to tempf */
 show();
}/* end of main */
```

(*a*)

FIGURE 12-30 Program with C and assembly language modules. (*a*) C mainline module. (*See also pp. 431–2.*)



```
; 8086 PROGRAM F12-30B.ASM

_TEXT           SEGMENT         BYTE PUBLIC 'CODE'
DGROUP          GROUP           _DATA,_BSS
                ASSUME          CS:_TEXT,DS:DGROUP,SS:DGROUP
_TEXT           ENDS

_DATA           SEGMENT WORD PUBLIC 'DATA'
S@      DB 'CELSIUS = %D, FAHRENHEIT = %D',0AH, 00H ; PRINTF STRING
_DATA           ENDS

_TEXT           SEGMENT         BYTE PUBLIC 'CODE'
PUBLIC _C2F
PUBLIC _SHOW
EXTRN _PRINTF:NEAR

_C2F            PROC            NEAR                    ; C2F function definition
                PUSH            BP                      ; Save old BP
                MOV             BP,SP                   ; Copy of SP to BP
                PUSH            SI                      ; Save SI
                MOV             AX,WORD PTR [BP+4]      ; Get TEMPC from stack
                MOV             DX,9
                MUL             DX
                MOV             BX,5
                CWD
                IDIV            BX
                MOV             SI,AX
                ADD             SI,32
                MOV             AX,SI                   ; Return value of F in AX
                POP             SI
                POP             BP                      ; Restore old BP
                RET
_C2F            ENDP

_SHOW PROC NEAR
                PUSH            WORD PTR DGROUP:_TEMPF ; Put value of TEMPF on stack
                PUSH            WORD PTR DGROUP:_TEMPC ; Put value of TEMPC on stack
                MOV             AX,OFFSET DGROUP:S@     ; Put offset of string on stack
                PUSH            AX
                CALL            NEAR PTR _PRINTF        ;
                ADD             SP,6                    ; Increment SP over arguments
                RET
_SHOW ENDP
_TEXT           ENDS

_BSS            SEGMENT WORD PUBLIC 'BSS'
                EXTRN _TEMPF:WORD
_BSS            ENDS

_DATA SEGMENT WORD PUBLIC 'DATA'
                EXTRN _TEMPC:WORD
_DATA ENDS
                END
```

(*b*)

FIGURE 12-30 (*Continued*) (*b*) Assembly language functions. (*See also next page.*)

```
DOSSEG
 .MODEL SMALL

 .CODE

PUBLIC _C2F
PUBLIC _SHOW
EXTRN _PRINTF:NEAR
_C2F          PROC          NEAR
          .
          .
          .
              RET
_C2F          ENDP

_SHOW PROC NEAR
          .
          .
          .
              RET
_SHOW ENDP

 .DATA
SA  DB 'CELSIUS = %D, FAHRENHEIT = %D'
    DB 0AH, 00H ; PRINTF STRING
    EXTRN _TEMPC:WORD

 .DATA?
              EXTRN _TEMPF:WORD

              END
```

(c)

FIGURE 12-30 (*Continued*) (c) Assembly language module using simplified segment directives.

pointer in a program intended to run on a PC type computer, because this is done by DOS and/or the C startup code. If you are writing a program for some other environment, you can declare a stack and initialize the stack pointer with a simple directive such as .STACK 200H.

We showed you the standard segment directive version of an assembly language module first, so that you could see how all the pieces fit together, but the simplified directives are obviously easier to use in your programs. Now that you know how to write C and assembly language modules that interface with each other, we will outline how you produce an executable program from these modules.

PRODUCING A .EXE FILE FOR MULTIMODULE PROGRAMS

If you are using the Turbo C + + environment, the steps in producing a .exe file from a multimodule program such as the one in Figure 12-30 are as follows. If you are using some other environment, the steps are very similar.

1. Create the C module using the editor. Don't forget any required extern directives.
2. Compile the module and repeat the edit-compile cycle until the compile is successful.
3. Create the assembly language module with the editor. Don't forget to include any required public and extrn directives. Save the module in a file with a .asm extension.
4. Press the Alt key and the spacebar to get to the menu containing Turbo Assembler. Move the highlighted box to the TurboAssembler line and press the Enter key.
5. Repeat the edit-assemble loop until the assemble is successful.
6. Go to the Project menu and select Open Project. When the dialog box appears, type in some appropriate name for your project and give it a .prj extension.
7. Use the Add Item line in the Project menu to add the name(s) of the C source (.C) files and the names of your assembly language object (.obj) files to the project file. Press the Esc key to get back to the project window.
8. Go to the Options menu and select Linker. In this menu go to the case sensitive link and press the Enter key to turn it off. TASM produces uppercase for all names, and this toggle will prevent link errors caused by uppercase/lowercase disagreements.
9. Go to the Compile menu, select build all, and press the Enter key. This tells the IDE tools to do a "make" on the files specified in the project list. Make checks the times and dates on the .obj files and the associated source files. If the times are different, the source modules are automatically recompiled. The resulting object files are linked with object files from .asm modules and object modules from libraries to produce the final .exe file.
10. Go to the Run menu and press the R key to run the program.

NOTE: For complex multimodule programs, you may want to use the separate tcc compiler and Tlink linker, which have some options not available in the integrated environment. We don't have space here to describe the operation of tcc.

In this chapter we used your knowledge of assembly language to quickly teach you about the C programming language. In the following chapters we will show you how C can be used for graphics, file handling, and communications programming.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

Integrated program development environment

Compiler optimizations

C language
- Variable types
- Variable declarations
- Simple pointers
- Array pointers
- Dereferencing a pointer
- Passing a parameter by value
- Passing a parameter by reference
- Preprocessor directives
- Assignment operator, =
- Arithmetic operators +, −, *, /, %, ++, −−
- Bitwise operators &, |, ˆ, ˜, <<, >>
- Combined operators
- Relational operators ==, !=, >, >=, <, <=
- Logical operators &&, | |, !
- Operator precedence
- If-else
- Switch and break statements
- Goto statement
- While and do-while loops
- For loops
- Function prototype, function declaration
- Function definition
- Function call
- Formal arguments
- Actual arguments
- Return statement
- Extern, automatic, static, and register storage classes
- Lifetime and visibility of variables
- Passing pointers to functions
- Pointers to functions
- Predefined library functions
- Turbo C++ memory models
- Cleaning up the stack
- Simplified segment directives

REVIEW QUESTIONS AND PROBLEMS

1. *a.* What is the index value for the first element in the cost array in Figure 12-1*a*?
b. Which element in the cost array is accessed by the term cost[index] during the second execution of the for loop?
c. What is the purpose of the #include<stdio.h> line at the top of the program in Figure 12-1*a*?
d. What does the word printf in the statement in Figure 12-1*a* refer to?

2. *a.* Describe the advantages of an integrated program environment such as the Turbo C++ IDE over the separate tools approach.
b. How does the IDE compiler let you know if it finds any errors when it compiles your program?
c. What is meant by the term watch in the IDE?

3. Give the range of values that can be represented by each of the following C data types.
a. Char
b. Int
c. Unsigned int
d. Long
e. Float

4. Write C declaration statements for each of the following variables:
a. An integer named total_boards.
b. A character named no, initialized with the ASCII code for lowercase n.
c. A floating-point variable named body_temp, initialized with 98.6.
d. A five-element integer array called scores.
e. A six-element integer array called scores and initialized with the values 95, 89, 84, 93, and 92 (last element uninitialized).
f. A pointer called ptr which points to the array declared in *e*.
g. A two-dimensional character array called screen which has 25 rows and 40 columns.
h. A three-dimensional character array called screen_buffer which has 4 pages of 25 rows and 80 columns.
i. An integer called monitor_start, initialized with +FE00H.
j. A character pointer named answer.
k. A pointer named ptr, initialized with the address of an integer variable called setpoint.
l. A pointer named wptr, initialized with the start of the array declared with the statement float net_weights[100];.

5. Describe the operation or sequence of operations performed by each of the following expressions:
a. 5−4*7/9
b. (a+4)*17−B/2+6
c. x+y++
d. x− −−Y
e. count +=4;
f. strobe_val & 0×0001
g. y=a>>4;
h. a=4;
b=39%a;
i. if (ch == 'Y' || ch == 'y')
goto start;

6. Write printf statements which
a. Print the decimal value of an integer named count.
b. Print a prompt message which tells the user to enter his or her weight.
c. Print the value of a float variable named conversion_factor with 4 decimal places and a total of 10 digits.
d. Print the value of a float variable called average_lunar_distance in exponential format.

7. Given the array declared by int nums[ ]={45, 65, 38, 72};, write a program which computes the average and prints the result.

8. Use Figure 12-12 to help you write a program which
Declares a six-element array of integers.
Reads five test scores entered by a user into the array.
Computes the average of the five scores and puts the computed average in the sixth element in the array.
Prints out the scores and the average with appropriate text.

9. Write a program which
Declares an array for 25 characters.
Prompts the user to enter his or her name.
Reads an entered name into the array.
Determines the number of characters in the name.
Prints out appropriate text and the number of letters.

10. *a.* Write a program section which calls the predefined exit function if the user enters a q or a Q on the keyboard.
b. The predefined character constant called EOF has a value of −1 (FFH). To produce this character on the keyboard, you hold the Ctrl key down and press the Z key. Write a program which
Declares an array for up to 1000 characters.
Reads characters from the keyboard and puts them in the array until the array is full or until the user enters an EOF character Ctrl Z.
Prints a "buffer full" message if 1000 characters entered.
Prints a "goodbye" message and exits to DOS if the EOF character is entered.

11. The character display on a CRT screen can be thought of as an array of 25 rows and 80 columns. Write a program which
Declares a character array of 25 rows and 80 columns.
Declares a character array initialized with your name.
Uses a nested for loop to write the ASCII code for a blank, 20H, to each element in the array.
Writes your name in the array elements which approximately correspond to the center of the screen.

12. Use the array-index method as shown in Figure 12-20*a* to write a program which
Declares a two-dimensional array of 7 rows and 3 columns.
Reads in the maximum temp and minimum temp for each of 7 days and puts the values in the array.
Computes the average temperature for each day and puts the result in the appropriate position in the third column of the array.
Computes the average maximum temperature for the week.
Computes the average minimum temperature for the week.
Computes the average temperature for the entire week.
Prints out the results with appropriate labeling.

13. Rewrite the program in problem 12 using pointer notation instead of array-index notation.

14. Explain the difference between formal arguments and actual arguments.

15. Write the declaration, definition, and call for a function which converts a Fahrenheit temperature to its Celsius equivalent. The formula is $F = 9C/5 + 32$.

16. Write a program which reads characters from the keyboard until an EOF (Ctrl Z) is entered, uses a function to detect and convert the ASCII codes for uppercase letters to their lowercase equivalents, and writes the codes in an array.

17. Given the array declared by int nums[ ]= 45,65,38,72;, write a function which computes the average of the four values and passes the average back to the calling program to print out.

18. Rewrite the answer to problem 12 so that it uses a function to compute the desired averages and print the result.

19. Give the lifetime and accessibility of each of the variables and functions declared here.
a. int scale_factor = 12;
b. char *text;
c. float tax(float income, float deductions);
d. static double debts;
main( )
{
e. static weight = 145;
f. register count = 23;
g. int tare;
}

20. Modify problem 11 to read in two sets of row, column coordinates from a user, store these values, and then call a function which ORs each element in the array between the specified coordinates with 80H.

21. What are the main advantages and the main disadvantages of using predefined C library functions?

22. Rewrite the Pythagoras program in Figure 12-27 so that it allows a user to enter values for side_a and side_b, does the computation, and sends the results to a printer.

23. What are the main points you have to consider when you want to write assembly language functions that will be called from a C program?

24. *a.* Name the six Turbo C + + memory models.
b. What are the main features which distinguish one memory model from another?
c. Describe the default memory model for the Turbo C + + compiler.
d. Show the simplified segment directives you would use for a small model assembly language module that contains only code and initialized variables.

25. Briefly describe the process used to develop a program which consists of assembly language modules and C modules.

26. Given the array declared with int screen[25][80];, write a C mainline which calls an assembly language function to write 20H in the low byte of each element and 07H in the high byte of each element.

CHAPTER 13

Microcomputer System Peripherals

In Chapter 11 we discussed the circuitry commonly found on the motherboard of a microcomputer. Included in this discussion was a section on the I/O connectors that allow you to plug in boards which interface with system peripherals. In this chapter we discuss the hardware and software of system peripherals such as keyboards, CRT displays, disk drives, printers, and speech I/O devices. Then in the next chapter we discuss serial data communication and network peripherals.

One important goal of this chapter is to help you understand the terminology of displays, disk drives, and printers so you feel comfortable with these when you read your BYTE magazine or when you walk into a computer store. Another important goal is to show you how to interface with displays, disk drives, and printers in your programs. For most of the examples in this and the following chapters we use IBM PC- and PS/2-type microcomputers.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Read and interpret data from the keyboard of an IBM PC- or PS/2-type microcomputer.
2. Describe the operation of basic hardware needed to produce raster scan text or graphics CRT displays.
3. Calculate the frequencies, frame buffer memory requirements, and memory access rate for a given-resolution raster scan display.
4. Describe how a video adapter such as a VGA displays 256 colors from a palette of 256K colors.
5. Use BIOS calls to display a text message on the CRT display of an IBM PC-compatible computer.
6. Use BIOS calls to produce graphics displays on the CRT display of an IBM PC-compatible computer.
7. Write simple C programs which use predefined functions to produce graphics displays on a CRT.
8. Describe how text and graphics displays are produced on large LCDs and plasma displays.
9. Show in general terms the formats in which digital data is stored on magnetic and optical disks.
10. Describe the operation of disk controller circuitry.
11. Use DOS function calls and C function calls to open, read, write, and close disk files.
12. Describe the print mechanism used in several common types of computer printers.
13. Describe how computer vision systems produce an image that can be stored in a digital memory.
14. Briefly describe how phoneme, formant filters, and linear predictive coding synthesizers produce human-sounding speech from a computer.
15. Briefly describe the basic principle used in speech-recognition systems.
16. Describe the operation and significance of a Digital Video Interactive system.

SYSTEM-LEVEL KEYBOARD INTERFACING

In Chapter 9 we discussed the tasks involved in getting meaningful data from a keyboard and in Figure 9-22 we showed you the hardware typically used to do these tasks in an IBM PC-type computer. Now we will show you how to read and interpret keyboard data in system-level programs.

When you press a key on an IBM PC-type computer, a type 9 interrupt is executed. The procedure for this interrupt reads the scan codes generated by the keyboard circuitry and determines the action to take, based on the code read. For certain special key combinations such as Shift-Print Screen or Ctrl-Alt-Del, the type 9 procedure will call other procedures to carry out the specified action. For standard keys the type 9 procedure will convert the scan codes to an ASCII equivalent code and put the ASCII code in a buffer. For special keys such as function keys and cursor-move keys the procedure generates *extended ASCII codes*.

To read the ASCII or extended ASCII codes from the buffer at the assembly language level, you use the BIOS INT 16H procedure. Perhaps you remember from the discussion of software interrupts in Chapter 8 that the ROMS in a microcomputer contain procedures for many input and output operations. Figure 8-9, for example, showed how you load some parameters in AH and AL

```
PARAMETERS FOR BIOS INT 16H KEYBOARD PROCEDURE

input:    AH = 0
function: Wait for next key pressed, return code
return:   key code in AL, scan code in AH

input:    AH = 1
function: Determine if character ready in buffer
return:   Zero flag = 1 - no character in buffer
          Zero flag = 0 - character in buffer

input:    AH = 2
function: Return status of Alt, Shift, Ctrl keys
return:   shift status in AL
```

FIGURE 13-1 Parameters for BIOS INT 16H BIOS procedure.

then execute the INT 17H instruction to send a character to a printer.

Figure 13-1 shows the format for the INT 16H procedure which you can use to read characters from the keyboard of an IBM PC- or PS/2-type computer. If you call the procedure with AH=0, execution will sit in a loop until a key is pressed. When a key is pressed, the procedure will return a value in AX. If the value returned in AL is not 0, then the value in AL is the ASCII code for the corresponding key or key combination shown in Figure 13-2*a*. Note that you can generate any desired hex value in AL by pressing the Alt key and the equivalent sequence of decimal digits on the numeric keypad. If the value returned in AL is zero, then the value in AH represents the extended ASCII code for one of the special keys or key combinations shown in Figure 13-2*b*.

If you call the INT 16H procedure with 1 in AH, the procedure will return with the carry flag set if there is no character in the buffer waiting to be read. If the

Value			
Hex	Dec	Symbol	Keystrokes
00	0	Blank (Null)	Ctrl 2
01	1	☺	Ctrl A
02	2	●	Ctrl B
03	3	♥	Ctrl C
04	4	♦	Ctrl D
05	5	♣	Ctrl E
06	6	♠	Ctrl F
07	7	•	Ctrl G
08	8	◘	Ctrl H, Backspace, Shift Backspace
09	9	○	Ctrl I
0A	10	█	Ctrl J, Ctrl ↵
0B	11	♂	Ctrl K
0C	12	♀	Ctrl L
0D	13	♪	Ctrl M, ↵, Shift ↵
0E	14	♫	Ctrl N
0F	15	☼	Ctrl O
10	16	►	Ctrl P
11	17	◄	Ctrl Q
12	18	↕	Ctrl R
13	19	!!	Ctrl S
14	20	¶	Ctrl T
15	21	§	Ctrl U
16	22	▬	Ctrl V
17	23	↨	Ctrl W

Value		
Hex	Dec	Keystrokes
18	24	Ctrl X
19	25	Ctrl Y
1A	26	Ctrl Z
1B	27	Ctrl [, Esc, Shift Esc, Ctrl Esc
1C	28	Ctrl \
1D	29	Ctrl]
1E	30	Ctrl 6
1F	31	Ctrl —
20	32	Space Bar, Shift, Space, Ctrl Space, Alt Space
21	33	!
22	34	"
23	35	#
24	36	$
25	37	%
26	38	&
27	39	'
28	40	(
29	41	)
2A	42	*
2B	43	+
2C	44	,
2D	45	–
2E	46	.
2F	47	/
30	48	0
31	49	1
32	50	2
33	51	3
34	52	4
35	53	5
36	54	6
37	55	7
38	56	8
39	57	9
3A	58	:
3B	59	;
3C	60	<
3D	61	=
3E	62	>
3F	63	?
40	64	@
41	65	A
42	66	B
43	67	C
44	68	D
45	69	E
46	70	F
47	71	G
48	72	H
49	73	I
4A	74	J
4B	75	K
4C	76	L
4D	77	M
4E	78	N
4F	79	O
50	80	P
51	81	Q
52	82	R
53	83	S
54	84	T
55	85	U
56	86	V
57	87	W
58	88	X
59	89	Y
5A	90	Z
5B	91	[
5C	92	\
5D	93	]
5E	94	^
5F	95	—
60	96	`
61	97	a
62	98	b
63	99	c
64	100	d
65	101	e
66	102	f
67	103	g
68	104	h
69	105	i
6A	106	j
6B	107	k
6C	108	l
6D	109	m
6E	110	n
6F	111	o
70	112	p
71	113	q
72	114	r
73	115	s
74	116	t
75	117	u
76	118	v
77	119	w
78	120	x
79	121	y
7A	122	z
7B	123	{
7C	124	¦
7D	125	}
7E	126	~
7F	127	Ctrl –

(*a*)

FIGURE 13-2 IBM PC keys and keycodes. (a) Standard key codes returned in AL. (*Continued on next page.*)

Second Code	Function
3	Nul Character
15	←
16–25	Alt Q, W, E, R, T, Y, U, I, O, P
30–38	Alt A, S, D, F, G, H, J, K, L
44–50	Alt Z, X, C, V, B, N, M
59–68	F1 to F10 Function Keys Base Case
71	Home
72	↑
73	Page Up and Home Cursor
75	←
77	→
79	End
80	↓
81	Page Down and Home Cursor
82	Ins(Insert)
83	Del(Delete)
84–93	F11 to F20 (Uppercase F1 to F10)
94–103	F21 to F30 (Ctrl F1 to F10)
104–113	F31 to F40 (Alt F1 to F10)
114	Ctrl PrtSc (Start/Stop Echo to Printer)
115	Ctrl ← (Reverse Word)
116	Ctrl → (Advance Word)
117	Ctrl End [Erase to End of Line (EOL)]
118	Ctrl PgDn [Erase to End of Screen (EOS)]
119	Ctrl Home (Clear Screen and Home)
120–131	Alt 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, = (Keys 2–13)
132	Ctrl PgUp (Top 25 Lines of Text and Home Cursor)

(*b*)

```
SHIFT STATUS BYTE RETURNED BY BIOS INT 16H

BIT  MEANING IF BIT IS A ONE

d0      Right shift key pressed

d1      Left shift key pressed

d2      Control key pressed

d3      Alt key pressed

d4      Scroll lock active

d5      Numeric lock active

D6      Caps lock active

D7      Insert state active
```

(*c*)

FIGURE 13-2 (*Continued*) (*b*) Extended ASCII codes returned in AH. (*c*) Status byte returned in AL with AH = 2 during call.

procedure returns with the carry flag = 0, the buffer contained a character, and that character has been read into AX as described before. This option allows you to check if a key has been pressed without having to sit in a loop until a key is pressed.

Finally, if you call the INT 16H procedure with a 2 in AH, the procedure will return the status of the Shift, Alt, and Ctrl keys, as shown in Figure 13-2*c*.

From the preceding discussion you can see that to interface with the keyboard from an assembly language program all you have to do is load the desired subfunction number (0, 1, or 2) in AH and execute the INT 16H instruction. The next question to answer is, How do you interface with the keyboard from a C program?

In the last chapter we showed you how to use predefined C functions such as scanf, getche, and gets to read characters from the keyboard. The problem with these functions is that they do not allow you to read anything but the standard ASCII codes (00–7FH). In many system programs you want to use the function keys, arrow keys, or other special keys to specify some course of action, so you need to be able to read in codes for these. The Turbo C + + run time libraries contain two predefined functions which you can call to read in key codes directly. Both of these use the BIOS INT 16H procedure.

The predefined function int bioskey (int cmd) will call the INT 16H procedure and pass it the subprocedure specified as cmd in the call. The statement key = bioskey(0);, for example, will wait until a key is pressed and assign the value returned in AX to key. You can then manipulate the value in key to determine which key was pressed.

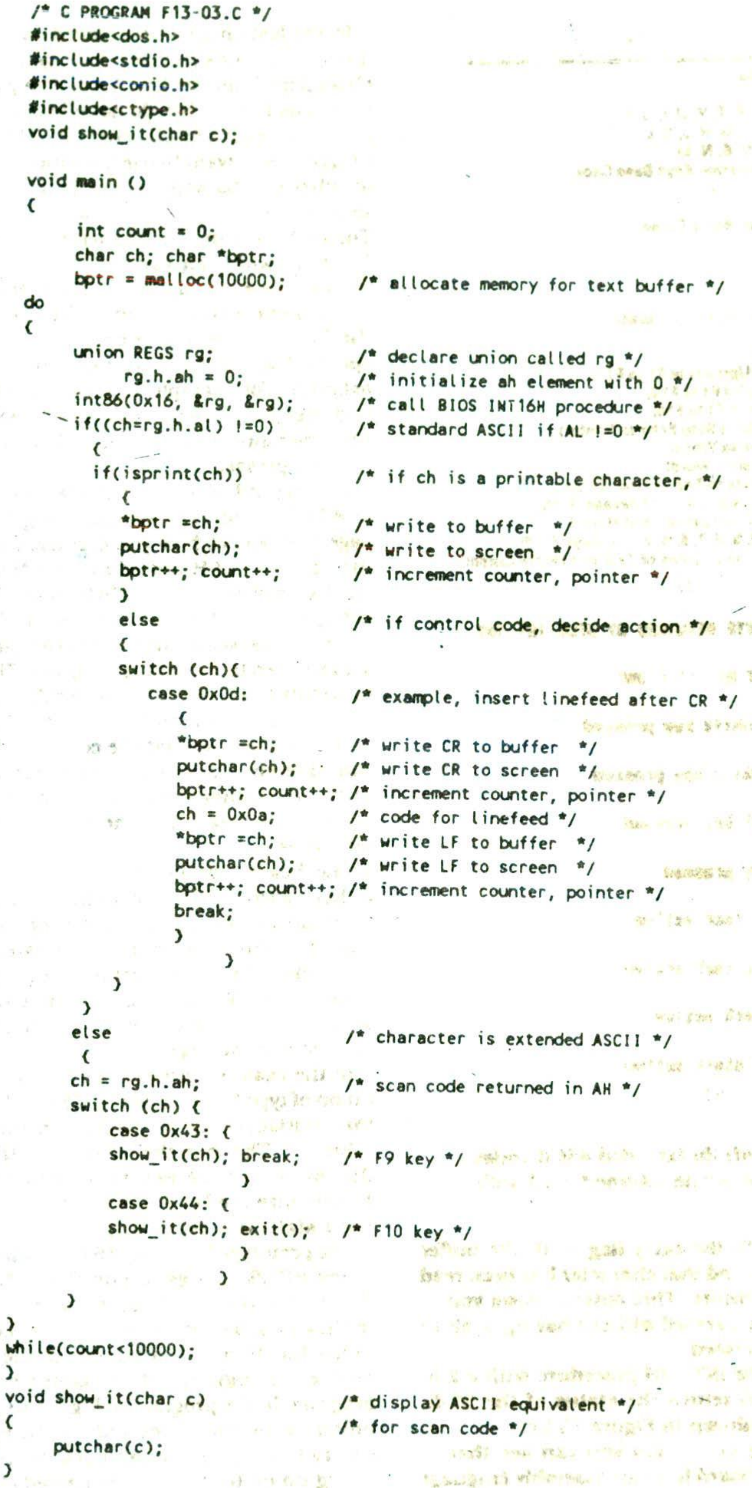
The second way to read the keyboard is with the int86() function. The example program in Figure 13-3, page 438, shows how you can use this function to call the BIOS INT 16H procedure, but this function can be used to call any of the BIOS procedures. The key to using this function is to understand how the register values are passed to the function and how register values are returned to the calling program. The technique used to do this is a C data structure called a *union*. In simple terms a union is a memory location assigned to two or more variables, so that the contents can be accessed in two different ways. You might, for example, create a union of an integer and an unsigned character so that you could access either the entire 16 bits or the two 8-bit halves.

The header file dos.h contains the prototype for a union called REGS. This union is composed of two structures which represent the register set of the 8086. One structure represents the registers as 8-bit values. The other structure represents the registers as 16-bit values. This allows you to initialize an 8-bit register or a 16-bit register and read a value from an 8-bit register or from a 16-bit register.

In the example program in Figure 13-3, we declare a union of type REGS called rg. The statement rg.h.ah = 0; then initializes the ah element in the structure with a value of 0. The .h in the reference to the union indicates that we want to access the structure of 8-bit registers. To initialize the DX register with a value of 0, we would use a statement such as rg.x.dx = 0;.

The prototype for the int86 function is int86(int intno, union REGS *inregs, union REGS *outregs). What all this means is that you pass the INT number, a pointer to the union which contains the register values to pass to the function, and a pointer to the union which will receive the register values passed back to the calling program. In the program in Figure 13-3 we use the same union, rg, for the inregs and the outregs. The statement ch = rg.h.al copies the value placed in the al element of the rg union to the variable named ch.

The rest of the example in Figure 13-3 shows you how to examine the value returned by INT 16H to determine the action to take. If the value returned in AL is not zero, then the code is in the range of 00–FFH. We use



```
/* C PROGRAM F13-03.C */
#include<dos.h>
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void show_it(char c);

void main ()
{
     int count = 0;
     char ch; char *bptr;
     bptr = malloc(10000);          /* allocate memory for text buffer */
do
{
     union REGS rg;                 /* declare union called rg */
          rg.h.ah = 0;              /* initialize ah element with 0 */
     int86(0x16, &rg, &rg);         /* call BIOS INT16H procedure */
     if((ch=rg.h.al) !=0)           /* standard ASCII if AL !=0 */
       {
       if(isprint(ch))              /* if ch is a printable character, */
          {
          *bptr =ch;                /* write to buffer  */
          putchar(ch);              /* write to screen  */
          bptr++; count++;          /* increment counter, pointer */
          }
          else                      /* if control code, decide action */
          {
          switch (ch){
             case 0x0d:             /* example, insert linefeed after CR */
                {
                *bptr =ch;          /* write CR to buffer  */
                putchar(ch);        /* write CR to screen  */
                bptr++; count++;    /* increment counter, pointer */
                ch = 0x0a;          /* code for linefeed */
                *bptr =ch;          /* write LF to buffer  */
                putchar(ch);        /* write LF to screen  */
                bptr++; count++;    /* increment counter, pointer */
                break;
                }
                    }
          }
       }
     else                           /* character is extended ASCII */
       {
     ch = rg.h.ah;                  /* scan code returned in AH */
     switch (ch) {
         case 0x43: {
         show_it(ch); break;        /* F9 key */
                    }
         case 0x44: {
         show_it(ch); exit();       /* F10 key */
                    }
                   }
     }
}
while(count<10000);
}
void show_it(char c)                /* display ASCII equivalent */
{                                   /* for scan code */
     putchar(c);
}
```

FIGURE 13-3 C program showing how to use int86( ) function call to read keyboard and decode the value read.

the predefined function isaprint() to determine if the code is a printable ASCII code, and if it is we write it to a buffer and send it to the screen. If it is not a printable code, we use a switch structure to determine what action to take. For the example here we showed you how to insert a linefeed character after a carriage return. You can add more case statements to perform the desired action for other special keys such as backspace.

If the value returned in AL is zero, the rg.h.ah element of the union will contain the scan code for the pressed key according to the values shown in Figure 13-2*b*. In the example in Figure 13-3 we show you how to use another switch structure to choose some action based on the code returned.

Now that you know more about reading characters from a microcomputer keyboard, let's dig into how character and graphics displays are produced.

MICROCOMPUTER DISPLAYS

Currently there are several different technologies used to display characters and graphics for a microcomputer. The most common type display is still the *cathode-ray tube* (CRT), so we will start the chapter with a discussion of the hardware and software for these displays. Later in the chapter we will discuss large *liquid-crystal displays* (LCDs) and plasma displays which are often used on laptop microcomputers.

Raster Scan Character Displays

RASTER SCAN BASICS

A CRT is basically a large, bottle-shaped vacuum tube. An electron gun at the rear of the tube produces a beam of electrons, which is directed toward the front of the tube by a high voltage. The inside surface of the front of the tube is coated with a phosphor substance which gives off light when it is struck by electrons. The color of the light given off is determined by the particular phosphor used.

The most common method of producing images on a CRT screen is to sweep the electron beam back and forth from left to right across the screen. When the beam reaches the right side of the screen, it is turned off (blanked) and retraced rapidly back to the left side of the screen to start over. If the beam is slowly swept from the top of the screen to the bottom of the screen as it is swept back and forth horizontally, the entire screen appears lighted. When the beam reaches the bottom of the screen, it is blanked and rapidly retraced back to the top to start over. A display produced in this way is referred to as a *raster scan* display. To produce an image, the electron beam is turned on or off as it sweeps across the screen. The trick here is to get the beam intensity or *video information* synchronized with the horizontal and vertical sweeping so the display is stable.

For a first example, Figure 13-4 shows the scanning used to produce pictures on a TV set and displays on some computer monitors. To get better picture resolution and avoid flicker, TVs use *interlaced scanning*. As shown in Figure 13-4*a*, this means the scan lines for one sweep of the beam from the top of the screen to bottom (field) are offset and interleaved with those of the next field. After every other field the scan lines repeat. Therefore, two fields are required to make a complete picture or frame. To give you some numbers for reference, black-and-white TVs in the United States use a horizontal sweep frequency of 15,750 Hz and a vertical sweep frequency of 60 Hz. Sixty fields per second are then swept out. Since each complete picture or frame

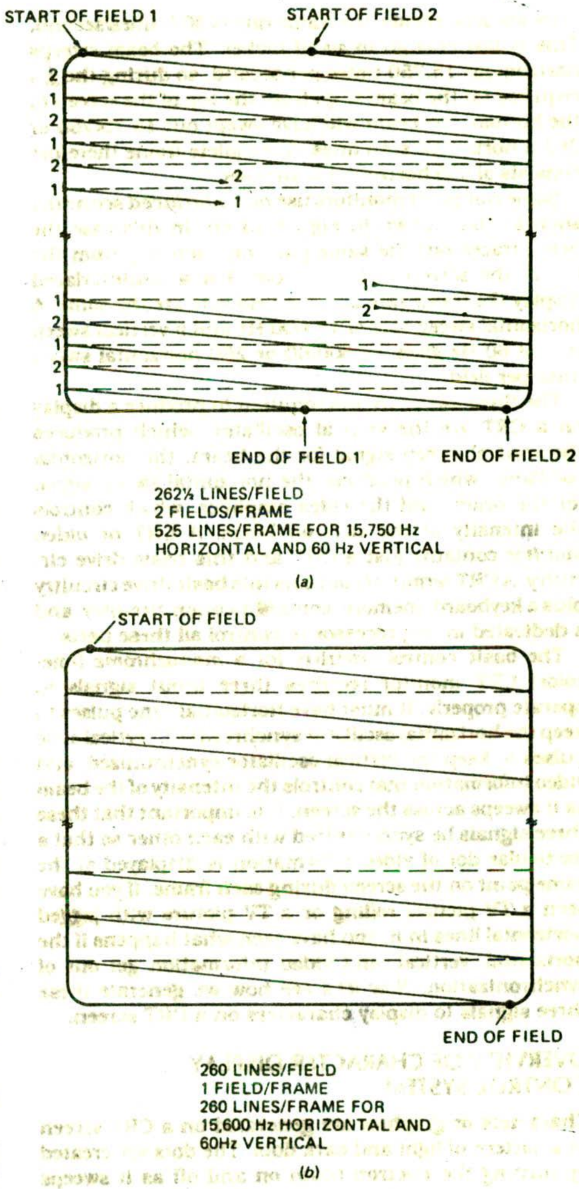


FIGURE 13-4 CRT scanning methods. (*a*) Interlaced. (*b*) Noninterlaced.

consists of two fields, the frame rate is 30 frames/second. This is fast enough to avoid flicker. The beam sweeps horizontally 15,750 times per second, so during the $\frac{1}{60}$ s required for the beam to go from the top of the screen to the bottom, the beam will have swept out 15,750/60 or 262.5 horizontal scan lines. A complete frame therefore consists of 525 horizontal scan lines.

Some computer monitors use *noninterlaced scanning* such as that shown in Figure 13-4*b*. In this case the beam traces out the same path on each trip from the top of the screen to the bottom. For a noninterlaced display the frame rate and the field rate are the same. A horizontal sweep rate of 15,600 Hz and a vertical sweep rate of 60 Hz gives 15,600/60 or 260 horizontal sweep lines per field.

The three basic circuits required to produce a display on a CRT are the vertical oscillator, which produces the vertical sweep signal for the beam; the horizontal oscillator, which produces the horizontal sweep signal for the beam; and the video amplifier, which controls the intensity of the electron beam. A *CRT* or *video monitor* contains just a CRT and this basic drive circuitry. A CRT *terminal* contains this basic drive circuitry plus a keyboard, memory, communication circuitry, and a dedicated microprocessor to control all these parts.

The basic control circuitry for a *monochrome* (one-color) CRT monitor requires three input signals to operate properly. It must have horizontal sync pulses to keep the horizontal oscillator synchronized, vertical sync pulses to keep the vertical oscillator synchronized, and video information that controls the intensity of the beam as it sweeps across the screen. It is important that these three signals be synchronized with each other so that a particular dot of video information is displayed at the same point on the screen during each frame. If you have seen a TV picture rolling or a TV picture with jagged horizontal lines in it, you have seen what happens if the horizontal, vertical, and video information get out of synchronization. Now let's see how we generate these three signals to display characters on a CRT screen.

OVERVIEW OF CHARACTER DISPLAY CONTROL SYSTEM

Characters or graphics are generated on a CRT screen as a pattern of light and dark dots. The dots are created by turning the electron beam on and off as it sweeps across the screen. Figure 13-5 shows how the letters P and H can be displayed in the upper-left corner of the screen in this way. The round dots in the figure represent the beam on, and the square boxes represent the beam off. As you can see, in this example the dot matrix for each character is 5 dots wide and 7 dots high. Other common dot-matrix sizes for character displays are 7 by 9, and 7 by 12, and 9 by 14.

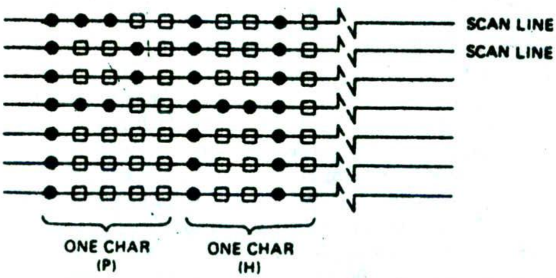


FIGURE 13-5 Producing a character display on a CRT screen with dots.

Figure 13-6 shows a block diagram of the circuitry needed to keep the pattern of dots for a page of text displayed on the screen of a CRT monitor. For this example assume that the display has 25 rows of characters with 80 characters per row.

The ASCII codes for the characters to be displayed on the screen are stored in a RAM. This RAM is often referred to as the *frame buffer* or the *display refresh RAM*. The RAM must contain at least one byte location for each character to be displayed. A display size of 25 rows with 80 characters in each row then requires 25 × 80 or about 2 Kbytes RAM. In an actual circuit this RAM is set up so that the microprocessor can access it to change the stored characters, or the display refresh circuitry can access it to keep the display refreshed on the screen.

The dot patterns for each scan line of each character to be displayed are stored in a ROM called a *character generator* ROM. Figure 13-7 shows the matrix for a typical character-generator ROM. This ROM uses a 7 by 9 matrix for the actual character, but the total dot space for each character is a 9 by 14 dot matrix. The extra dots are included to leave space between characters and between rows of characters. Also, the extra space allows lowercase letters to be dropped in the matrix so that descenders are shown correctly. Each dot row in Figure 13-7 represents the pattern of dots for a horizontal scan line of the character.

To start the display in the upper left corner, the character counter and the character row counter outputs are all 0's so the ASCII code for the first character in the display RAM is addressed. The ASCII code from the addressed location is output by the RAM to the address inputs of the character-generator ROM. These inputs essentially tell the character generator which character is to be displayed.

To keep track of which line in a character row is currently being swept out, we use a scan line counter. For our example here each row of characters has 14 dot rows or scan lines, so the scan line counter is a modulo-14 counter. The outputs of this counter are connected to four additional address inputs on the character generator ROM.

Given an ASCII code and a dot row count, the character-generator ROM will output the 9-bit dot pattern for one dot row in the character. For the first scan across the screen, the counter will output 0000, so the dot pattern output will be that for dot row 0000 of the character.

The output from the character generator is in parallel form. In order to turn the beam on and off at the correct time as it sweeps across the screen, this dot pattern must be converted to serial form with a parallel-in, serial-out shift register. The high-frequency clock used to clock this shift register is called the *dot clock* because it

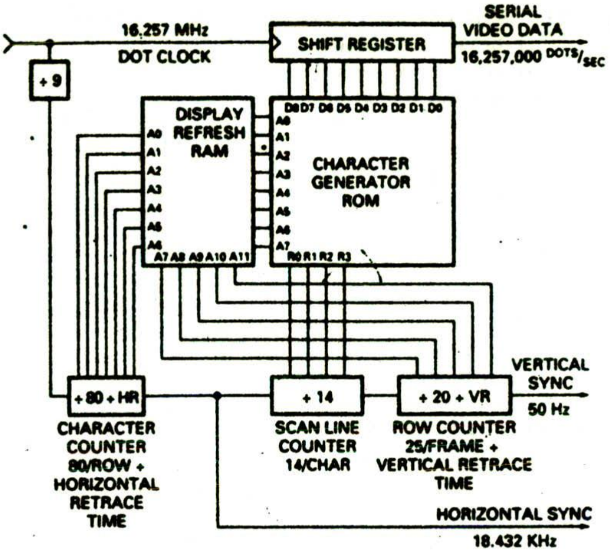


FIGURE 13-6 Block diagram of circuitry to produce dot-matrix character display on CRT.

controls the rate at which dot information is sent out to the video amplifier. As you can see in Figure 13-6, we used a dot clock frequency of 16.257 MHz for this example.

After the nine dots for the first scan line of the first character are shifted out, the character counter is incremented by 1. The outputs of the character counter are connected to some of the address inputs of the display refresh RAM, so when this count is incremented, the ASCII code for the next character in the top row is addressed in the refresh RAM. The ASCII code for this second character will be output to the character-generator ROM. Since the dot line counter inputs to the ROM are still 0000, the ROM will output the 9-bit dot pattern for the top scan line of the second character in the top row of characters on the screen. When all the dots for the top scan line of this character are all shifted out, the character counter will be incremented by 1 again, and the process will be repeated for the third character in the top row of characters. The process continues until the first scan line for all 80 characters in the top row of characters is traced out.

A horizontal sync pulse is then produced to cause the beam to sweep back to the left side of the screen. After the beam retraces to the left, the character counter is rolled back to zero to point to the ASCII code for the first

DOT ROW									
0000	0	0	0	0	0	0	0	0	0
0001	0	●	●	●	●	●	●	0	0
0010	0	●	0	0	0	0	0	●	0
0011	0	●	0	0	0	0	0	●	0
0100	0	●	0	0	0	0	0	●	0
0101	0	●	●	●	●	●	●	0	0
0110	0	●	0	0	0	0	0	0	0
0111	0	●	0	0	0	0	0	0	0
1000	0	●	0	0	0	0	0	0	0
1001	0	●	0	0	0	0	0	0	0
1010	0	0	0	0	0	0	0	0	0
1011	0	0	0	0	0	0	0	0	0
1100	0	0	0	0	0	0	0	0	0
1101	0	0	0	0	0	0	0	0	0

CAPITAL OR UPPERCASE

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	●	0	●	●	●	0	0	0
0	●	●	0	0	0	●	0	0
0	●	0	0	0	0	●	0	0
0	●	0	0	0	0	●	0	0
0	●	●	0	0	0	●	0	0
0	●	0	●	●	●	0	0	0
0	●	0	0	0	0	0	0	0
0	●	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

SMALL OR LOWERCASE

FIGURE 13-7 Dot matrix for 9 × 14 character-generator ROM.

character in the row again. The dot line counter (R0–R3) is incremented to 0001 so that the character generator will now output the dot patterns for the second scan line of each character. After the dot pattern for the second scan line of the first character in the row is shifted out to the video amplifier, the character counter is incremented to point to the ASCII code for the second character in the display RAM. The process repeats until all the scan lines for one row of characters have been scanned.

The character row counter is then incremented by 1. The outputs of the character counter and the character row counter now point to the display RAM address where the ASCII code for the first character of the second row of characters is stored. The process we described for the first row will be repeated for the second row of characters. After the second row of characters is swept out, the process will go on to the third row of characters, and then on to the fourth, and so on until all 25 rows of characters have been swept out.

When all the character rows have been swept out, the beam is at the lower right corner of the screen. The counter circuitry then sends out a horizontal sync pulse to retrace the beam to the left side of the screen and a vertical sync pulse to retrace the beam to the top of the screen. When the beam reaches the top left corner of the screen, the whole *screen-refresh* process that we have described will repeat. As we mentioned before, the entire screen must be scanned (refreshed) 30 to 75 times a second to avoid a blinking display. For the example in Figure 13-6 we used 50 Hz for the frame-refresh rate. Now let's look at a simple example of an actual CRT controller.

THE IBM PC MONOCHROME ADAPTER

Figure 13-8 shows a block diagram for the IBM PC monochrome display adapter board. This adapter is somewhat obsolete, but it is a good next step from the generic circuit in Figure 13-6. Take a look at Figure 13-8 and see what parts you recognize from our previous discussions. You should quickly find the CRT controller, character generator, and dot shift register. Next, find the 2-Kbyte memory where the ASCII codes for the characters to be displayed are stored. To the right of this memory is another 2-Kbyte memory used to store an *attribute* code for each character. An attribute code specifies whether the character is to be displayed normally, with an underline, with increased or decreased intensity, blinking, etc. You may have observed, for example, that it is common practice to display a screen menu at reduced intensity so it does not distract from the main text on the screen. For future reference Figure 13-9 shows the meaning of the bits in the attribute byte. If the B bit is a 1, the displayed character will blink. If the I bit is a 1, the character will be highlighted; in other words, it will have increased intensity. These bits give you several choices for how you want each character displayed on the screen.

As we discussed in a preceding section, ASCII codes from the display refresh RAM go to the character generator. Also going to the character generator are four address lines which specify the dot line of the character scanned. The counter which generates this address is contained in the MC6845 CRT controller device. The output from the character generator goes to a shift register to be converted to serial form for the video amplifier. The shift register is clocked by the 16.257-MHz dot clock. Circuitry in the video process logic section divides this dot clock signal by 9 to produce the character clock signal of 1,787,904 Hz. This character clock signal pulses each time a new ASCII character needs to be fetched from the display refresh RAM and a new attribute from the attribute RAM.

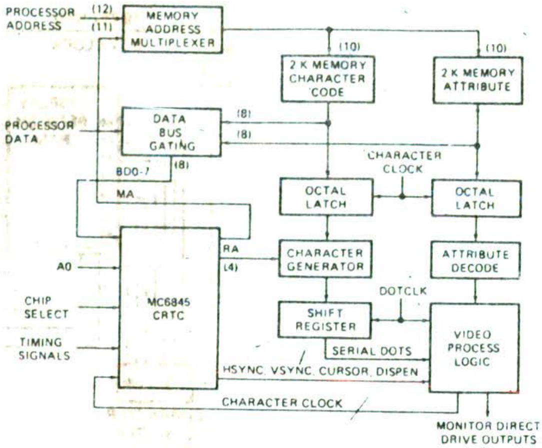
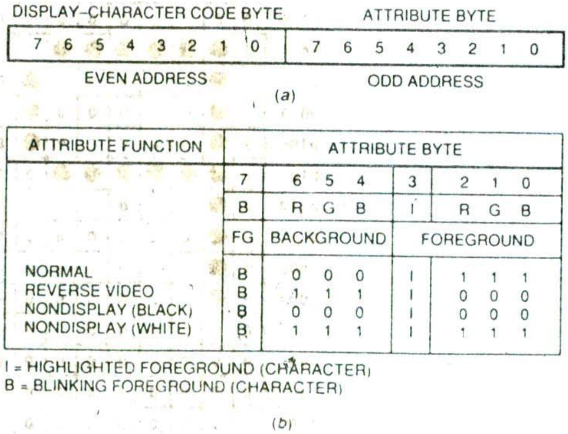


FIGURE 13-8 Block diagram of IBM PC monochrome adapter board.

Next observe that there is a multiplexer in series with the address lines going to the character and attribute memories. This multiplexer is connected so that either the CPU or the CRT controller can access the display refresh RAM.

To keep the display refreshed, the 6845 CRT controller device sends out the memory address for a character code and an attribute code. The character clock signal



ATTRIBUTE FUNCTION	ATTRIBUTE BYTE							
	7	6	5	4	3	2	1	0
	B	R	G	B	I	R	G	B
	FG	BACKGROUND			FOREGROUND			
NORMAL	B	0	0	0	I	1	1	1
REVERSE VIDEO	B	1	1	1	I	0	0	0
NONDISPLAY (BLACK)	B	0	0	0	I	0	0	0
NONDISPLAY (WHITE)	B	1	1	1	I	1	1	1

I = HIGHLIGHTED FOREGROUND (CHARACTER)
B = BLINKING FOREGROUND (CHARACTER)

(b)

FIGURE 13-9 Data storage format for IBM PC character displays. (*a*) Character byte and attribute byte in word. (*b*) Attribute byte format.

latches the code from memory for the character generator and the attribute code for the attribute decode circuitry. The character clock also increments the address counter in the 6845 to point to the next character code in memory. The next character clock transfers the next codes to the character generator and attribute decoder. The process cycles through all of the characters on the page and then repeats.

Now, when you want to display some new characters on the screen, you simply have the CPU execute some instructions which write the ASCII codes for the new characters to the appropriate address in the display RAM. When the address decoding circuitry detects a display RAM address, it produces a signal which toggles the multiplexers so that the CPU has access to the display RAM. The question that probably occurs to you at this point is, What happens if the 6845 and the CPU both want to access the display RAM at the same time? There are several solutions to this problem. One solution is to allow the CPU to access the RAM only during horizontal and/or vertical retrace times. Another solution is to interleave 6845 accesses and CPU accesses. This is how it is done on the IBM monochrome board.

The CPU is allowed to access the RAM during one-half of the character clock signal and the 6845 is allowed to access the RAM during the other half of the character clock signal. If the CPU tries to access the display RAM during the controller's half of the character clock cycle, a not-ready signal from the CRT controller board will cause the processor to insert WAIT states until the half of the character clock signal when it can access the display refresh RAM.

The 6845 CRT controller in Figure 13-8 contains the chain of counters shown in Figure 13-6 and other circuitry needed to produce horizontal blanking pulses, vertical blanking pulses, a cursor, scrolling, and highlighting for a CRT display. Several manufacturers offer CRT controller ICs that contain different amounts of the required circuitry. The Motorola MC6845 is used in both the monochrome and the color/graphics adapter boards for the IBM PC.

CRT DISPLAY TIMING AND FREQUENCIES

There are many different horizontal, vertical, and dot clock frequencies commonly used in raster scan CRT displays. The horizontal sweep frequency is usually in the range of 15 to 50 kHz, the vertical sweep frequency is usually 50 or 60 Hz, and the dot clock frequency is usually in the range of 10 to 100 MHz. As a first example, let's look a little closer at the frequencies used in the IBM PC monochrome adapter we discussed in the preceding section.

To refresh your memory, the IBM PC monochrome display adapter produces a display of 25 rows of 80 characters/row. Each character is produced as a 7 by 9 matrix of dots in a 9 by 14 dot space. This means that because clear space is left around each actual character, each character actually uses 9 dot spaces horizontally and 14 scan lines vertically.

The active horizontal display area then is 9 dots/character × 80 characters/line or 720 dots per line. The active vertical display area is 25 rows × 14 scan lines/row or 350 scan lines.

Now, according to the IBM Technical Reference Manual, the monochrome adapter uses a dot clock frequency of 16.257 MHz. This means that the video shift register is shifting out 16,257,000 dots/second. The manual also indicates that the board uses a horizontal sweep frequency of 18,432 lines/second. Dividing 16,257,000 dots per second by 18,432 lines per second tells you that the board is shifting out 882 dots/line. Just above we showed you that the active display area of a line is only 720 dots. The extra 162 dot times actually present give the beam time to get from the right edge of the active display to the right edge of the screen, retrace to the left edge of the screen, and sweep to the left edge of the active display area. The large number of extra dot times is necessary because most monitors have a large amount of *overscan*. Overscan means that the beam is actually swept far off the left and right sides of the screen. This is done so that the portion of the sweep actually displaying the characters is linear and the characters do not run off the edges of the screen.

The manual for the monochrome display adapter also indicates that the frame rate is 50 Hz. In other words, the beam sweeps from the top of the screen to the bottom and back again 50 times/s. To see how many horizontal lines are in each frame, you can divide the 18,432 lines/s by 50 frames/s to give 369 scan lines/frame. As we showed before, the active vertical display area is 350 lines. The 19 extra scan line times give the beam time to get to the bottom of the screen, retrace to the top of the screen, and get to the start of the active display area again.

Another point it is appropriate to mention here concerns the bandwidth required by the video amplifier in the monitor. In order to produce a sharp display the video amplifier in the monitor must be able to turn on and off fast enough so that dots and undots don't smear together. For our example here, the dot clock frequency is 16.257 MHz. This means that the dot shift register is shifting out 16,257,000 dots/s. If alternating dots and undots are being shifted out, then the waveform on the serial output pin of the shift register will be a square wave with a frequency of half that of the dot clock or 8.1285 MHz. In order to produce a clear display with this many dots per line, then, the video amplifier in the monitor connected to the display adapter must have a bandwidth of at least 8 MHz.

This bandwidth requirement is the reason that normal TV sets connected to computers cannot display high-resolution 80-character lines for word processing, etc. In order to filter out the 4.5-MHz sound subcarrier signal and the 3.58-MHz color subcarrier, the bandwidth of TV video amplifiers is limited to 3 MHz or less.

A final point we want to make about CRT timing is how often the display-refresh RAM has to be accessed. As the circuitry scans one line of the display, it has to access a new character in RAM after each 9 dots are shifted out, assuming 9 dots horizontally per character. Dividing the dot clock frequency of 16,257,000 dots per second by 9 dots/character tells you that characters are read from RAM at a rate of about 1,806,333 characters/

s, or one character every 553 ns! As we discussed in the last section, the CRT controller device accesses the display refresh RAM during one half of this time, and the microprocessor accesses the frame buffer RAM during the other half of this time. Only about 200 ns are actually available for access to the RAM during each half of the character clock time. As we show later, higher resolution and color displays require even faster memory access.

Raster Scan Graphics Displays

MONOCHROME GRAPHICS

As we discussed previously, characters can be displayed on a CRT screen by sending out a series of dots and undots to the video amplifier. The ASCII codes for the characters to be displayed are stored in a display-refresh RAM. As shown in Figure 13-6, the character-generator ROM uses an ASCII code from RAM and a 4-bit code from the dot row counter to produce the dot pattern for the specified scan line in the character.

Now, suppose that the character generator is left out of this circuit and the outputs of the RAM are connected directly to the inputs of an 8-bit dot shift register. And further suppose that instead of storing the ASCII codes for characters in the RAM, we store in successive memory locations the dot patterns we want for each 8 dots of a scan line.

When a byte is read from the RAM and loaded into the shift register, the stored dot pattern will be shifted out to the CRT beam to produce the desired pattern for 8 dots along a section of a scan line on the screen. When the next RAM byte is transferred to the shift register, it will produce the next 8 dots along the scan line. The process is continued until all the dot positions on the screen have been refreshed. The entire screen then can be thought of as a matrix of dots. Each dot can be programmed to be on or off by putting a 1 or a 0 in the corresponding bit location in RAM. A graphics display produced in this way is known as a *bit-mapped raster scan display*. Each dot or, in some cases, block of dots on the screen is called a *picture element*. Most people shorten this to *pixel* or *pel*. For our discussions here let's assume a pixel is 1 dot.

Now, suppose that we want a monochrome graphics display of 640 pels horizontally by 200 pels vertically. This gives a total of 200 × 640 or 128,000 dots on the screen. Since each dot corresponds to a bit location in memory, this means that we have to have at least 128,000 bits or 16 Kbytes of RAM to hold the pel information for just one display screen. Compare this with the 4 Kbytes needed to hold the ASCII codes and attributes for an 80 by 25 character display. As we will show you a little later, producing a color graphics display with a large number of pels requires even more memory.

Monochrome graphics displays get boring after a while, so let's see how you can get some color in the picture.

COLOR MONITORS AND COLOR GRAPHICS

The screen of a monochrome CRT is coated with a single type phosphor, which produces a color specific to that phosphor when bombarded with electrons from the single electron gun at the rear of the tube. To produce a color CRT display, we apply dots or bars of red, green, and blue phosphors to the inside of the CRT. One very common approach is to have dots of the three phosphors in a line pattern as shown in Figure 13-10. The dots are close enough together so that to your eye they appear as a single dot or pixel. Three separate electron beams are used to bombard the three different phosphors. A "shadow mask" just behind the screen of the CRT helps prevent electrons intended for one color phosphor from falling on the other color phosphors. The distance between the holes in the shadow mask of a color CRT or the distance between pixels on the screen is referred to as its *pitch*. The pitch of commonly available CRTs is in the range of 0.21 mm to 0.66 mm. Smaller pitch and smaller dots mean that more dots can be put on the screen and therefore the screen has better resolution. The trade-off, however, is that as the pitch and dot size are made smaller, the beam current must be increased to get acceptable intensity. A large percentage of the beam current hits the shadow mask, and if this current is too high, it may overheat and warp the shadow mask and permanently distort the image on the screen.

A CRT monitor designed to produce color displays is commonly referred to as an *RGB monitor* or an *RGBI monitor*. In addition to red, green, and blue signal inputs, an RGB monitor has a horizontal sync input, a vertical sync input, and—in some cases—an intensity input. An RGB monitor must be designed to work with the display format and sync frequencies of the circuitry in the microcomputer. Fortunately, some monitors such as the NEC MultiSync, the Sony Multiscan, and the Magnavox MVX9CM will work correctly with a wide range of display formats and sync frequencies.

The apparent color of a pixel to your eye is determined by the intensity ratio of the three electron beams and the total intensity of the three beams. Figure 13-11

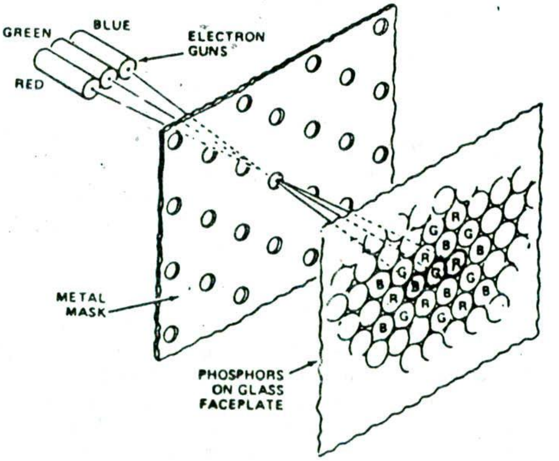


FIGURE 13-10 Three-color phosphor dot pattern used to produce color pixels on a CRT screen.

I	R	G	B	COLOR
0	0	0	0	BLACK
0	0	0	1	BLUE
0	0	1	0	GREEN
0	0	1	1	CYAN
0	1	0	0	RED
0	1	0	1	MAGENTA
0	1	1	0	BROWN
0	1	1	1	WHITE
1	0	0	0	GRAY
1	0	0	1	LIGHT BLUE
1	0	1	0	LIGHT GREEN
1	0	1	1	LIGHT CYAN
1	1	0	0	LIGHT RED
1	1	0	1	LIGHT MAGENTA
1	1	1	0	YELLOW
1	1	1	1	HIGH INTENSITY WHITE

FIGURE 13-11 Sixteen colors produced by different on and off combinations of red, green, and blue beams at normal and increased intensity.

shows 16 colors that can be produced when different combinations of on and off signals are applied to the three beams and to an overall intensity input. A 1 in the I bit means that the overall intensity of the beam is increased to lighten the color, as shown. If all three beams are off, the dot is, of course, black. If the beams are all turned on, then the dot will appear white. Other combinations give the other 14 shades shown.

To give a greater range of colors, newer monitors are designed to accept analog RGB signals instead of just digital RGB signals. The signals for these analog inputs are produced with D/A converters. Using a 2-bit D/A converter to produce each color signal, for example, gives 4 × 4 × 4 or 64 colors. However, as we discuss in the next section, increasing the number of colors increases the amount of memory needed in the frame buffer and the rate at which the memory must be accessed.

PALETTES, PIXEL PLANES, AND VRAMS

For a monochrome graphics display, the data for each pixel is stored in a single bit in the display-refresh RAM. Color displays require more than 1 bit per pixel, because the red, green, and blue data for each pixel must be stored. For example, 2 bits are required to specify one of 4 colors, 3 bits are required to specify one of 8 colors, 8 bits are required to specify one of 256 colors, etc. The number of colors we want to produce on a display then has a direct impact on the amount of memory required for the frame buffer. As an example of this, suppose that we want a 640 × 480 pixel display with 256 colors on an 8086 system. The total number of pixels in the display is 640 × 480, or 307,200. To specify one of 256 colors, 8 bits (1 byte) are required for each pixel. The total amount of frame buffer memory needed then is 307,200 bytes. Aside from the cost, this is an excessive amount of memory to devote to the display in a system that can address a total of only 1 Mbyte of memory.

To reduce the number of bits required for storing pixel data and still be able to display a wide range of colors, we use a *palette* scheme. The term palette is used here in about the same way an artist uses the term. An artist's palette holds the paint colors that he or she has available. The artist, for example, may have 16 colors on the palette, but for a particular painting he or she may use only 4 of the colors. We might say, then, that the artist has chosen 4 colors from a palette of 16.

As a first graphics example of this, the IBM Color Graphics Adapter board (CGA) can display medium resolution (320 × 200 pixel) graphics with 4 colors from a palette of 16 colors. Since only 4 colors are used at a time, only 2 bits of memory are required to hold the data for each pixel.

As a second example of a graphics palette, we might for our 640 × 480 pixel system decide to display 16 colors from a palette of 256, instead of all 256 colors. Only 4 bits are required to store the pixel data for 1 of 16 colors, so the frame buffer can be half the size it would be for a direct 256-color display. A little later we will take a look at how different systems implement the palette approach in hardware.

Another limiting factor in the design of a high-resolution color graphics system is the rate at which pixel data can be read from the frame buffer. For example, suppose that we want a 640 × 480 pixel display with 256 colors. As we said before, this requires 1 byte of memory per pixel, or a total of 307,200 bytes of memory. Assuming a frame rate of 50 Hz, each byte would have to be read from memory 50 times per second. This corresponds to 15,360,000 bytes per second, or 1 byte every 65 ns. (The time is actually shorter than this because all 307,200 accesses must occur during the active display time.) As we explained in Chapter 11, the read cycle times for common DRAMs is considerably longer than 65 ns. This means that we can't use DRAMs for the frame buffer unless we can find some way to allow more time for each access. There are several ways to do this.

The first step we take to give more time for the refresh controller access to the frame buffer is to allow the microprocessor to access the buffer only during horizontal and vertical retrace times. This gives all the time between characters or pixels to the controller instead of timesharing as we described for the monochrome adapter board.

A second way to reduce the required memory access rate is to use the palette scheme to reduce the number of bits required to store the data for each pixel. As we showed before, reducing a display to 16 colors from a palette of 256 instead of a direct 256-color display cuts the size of the display memory in half. Since only 4 bits are required to specify one of 16 colors, the data for 4 pixels can be packed in a single word, as shown in Figure 13-12*a*, page 446. Each memory access then reads in the data for 4 pixels.

A third method of reducing the access rate for the frame buffer is to set the memory up as parallel planes. Figure 13-12*b* attempts to show this in diagram form for a system which requires 4 bits per pixel. As you can

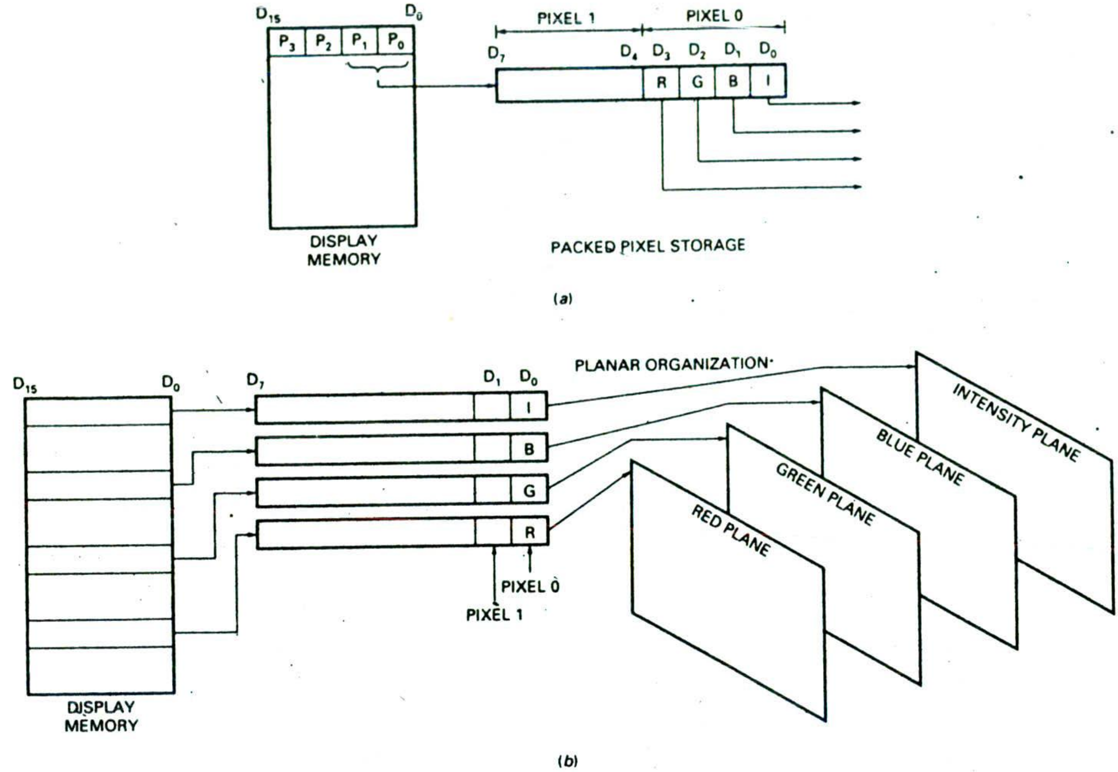


FIGURE 13-12 Frame buffer memory configurations. (*a*) Packed pixel. (*b*) Planar.

see, the 4 data bits for a pixel are stored at the same bit position in four different memory locations. When the controller transfers a word from each of the four memory locations to its internal registers, it has all the data it needs for 16 pixels. During the time that these 16 pixels are being swept out, the DRAMs will recover and can be accessed again. Additional planes can be added in parallel to store more bits per pixel.

Still another method of solving the memory access rate problem is to build the frame buffer with special DRAMs called *video RAMs* or *VRAMs*. Figure 13-13 shows a block diagram of the TI TMS44C251 VRAM. The DRAM section of the device consists of four arrays, which each store 256 Kbits. To the DRAM controller and the microprocessor circuitry, this device functions as a 256K × 4 device for read and write operations. The DRAM controller will supply RAS, CAS, multiplexed address, and refresh signals to it just as it would to any other DRAM.

To output data to a video controller, however, the 512 bits stored in each row in a DRAM array are transferred in parallel to a 512-bit register. The outputs of the register are connected to a 512-input multiplexer, which routes one of the register outputs to an SDQ output. As the multiplexer is stepped through its 512 positions, it outputs the 512 data bits one after the other to the SDQ output. The point here is that a VRAM can rapidly shift out the data for 512 pixels. A TMS44C251, for example, can shift out bits at up to 33 MHz, which is more than fast enough for a 800 × 512 pixel display. (Not counting overscan, this calculation is: (800 × 512 pixels/frame) × 60 frames/s = 24,576,000 pixels/s).

VRAMs can be used to store data in packed pixel

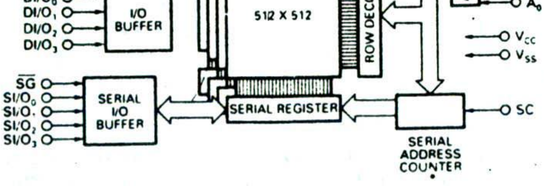


FIGURE 13-13 Block diagram of TI TMS44C251 video RAM (VRAM). (*Courtesy Texas Instruments Inc.*)

format or in planar format. Probably the easiest format to visualize is the planar. A single TMS44C251 can store four complete bit planes for a 512 × 512 pixel display. The devices can be cascaded to give scan lines longer than 512 pixels, or more than 4 bit planes. VRAMS are somewhat more expensive than standard DRAMS, but they make it relatively easy to implement a high-resolution display.

Now that you have a general awareness of how color graphics are produced, let's look at some specific examples.

Common Microcomputer Display Formats and Hardware

INTRODUCTION

There are an almost unbelievably large number of hardware configurations and formats for microcomputer displays. Figure 13-14 is an attempt to show the major display formats available on various IBM-type microcomputers. Reading the table from top to bottom essentially traces the development steps for IBM personal computer graphics during the last 10 years.

The IBM PC, the PC/XT, and the PC/AT do not have built-in graphics capability. For these computers you choose the text/graphics capability you want, buy the appropriate adapter board, plug it into one of the I/O slots in the motherboard, and connect a compatible monitor.

As we described in the preceding section, the IBM monochrome adapter board produces an 80 × 25 character display, but it does not produce graphics. An improvement on the basic monochrome adapter was the Hercules, Inc. monochrome adapter, which can display monochrome text or graphics in a 720 × 348 pixel format.

ADAPTER	MODE	RESOLUTION	COLORS	SIGNAL	COMPUTERS
			PALETTE		
CGA	ALPHA	25x80	4/16	DIGITAL	PC,XT,AT
	LOW RES	160x100	4/16	DIGITAL	PC,XT,AT
	MED RES	320x200	4/16	DIGITAL	PC,XT,AT
	HI RES	640x400	2/16	DIGITAL	PC,XT,AT
HERCULES	MONO	720x348	2	DIGITAL	PC,XT,AT
	COLOR	720x348	16/64	DIGITAL	PC,XT,AT
EGA		640x350	16/64	DIGITAL	PC,XT,AT
MGA		320x200	256	ANALOG	PS2-25,30
		640x480	2	ANALOG	PS2-25,30
VGA	11H	640x480	2	ANALOG	PS2-50,80
	12H	640x480	16/256K	ANALOG	
	13H	640x200	256/256K	ANALOG	
SUPER VGA		640x480	256/256K	ANALOG	ADAPTER
8514/A		1024x768	256/256K	ANALOG	ADAPTER

FIGURE 13-14 Major display formats available on various IBM-type microcomputers.

The most commonly used graphics format on the IBM Color Graphics Adapter (CGA) adapter is the medium-resolution (320 × 200 pixel) graphics mode, which displays 4 colors from a palette of 16 colors. A CGA adapter also has an alphanumeric or character mode, but in this mode it uses only an 8 × 8 dot matrix for each character. This makes its text display unpleasant to look at for long periods of time.

To solve this problem IBM developed the Enhanced Graphics Adapter (EGA) board. The EGA board has 25 × 80 text mode, which uses an 8 × 14 dot matrix for characters so text is more readable than that produced by a CGA card. The EGA board can operate in the CGA graphics modes and other graphic modes such as a 640 × 350 display with 16 colors from a palette of 64. To be able to display all the 64 colors in this mode, the monitor used must have a red-intensified input, a green-intensified input, and a blue-intensified input in addition to the standard red, green, and blue inputs.

Further improvement came with the IBM PS/2 line of microcomputers, which have CRT controllers included on their motherboards. The Multicolor Graphics Array (MCGA) found on the PS/2 Models 25 and 30 gives these machines all the display modes of an EGA and several others. Among the additional display modes are a 320 × 200 pixel display mode with 256 colors and a 640 × 480 pixel two-color display mode.

In the PS/2 models 50, 60, 70, and 80, a video graphics array (VGA) device produces a wide variety of display modes. In addition to EGA-compatible modes, a VGA has a 640 × 480 graphics display with 16 colors from a palette of 256K, a 320 × 200 graphics mode with 256 colors from a palette of 256K, and 25 × 80 text mode which uses an 8 × 16 dot matrix for characters.

To achieve 256 colors the MCGA and VGA circuits generate analog red, green, and blue signals instead of the digital RGB signals produced by EGA and earlier display adapters. The monitor for an MCGA or VGA system must be able to accept the analog color signals. Incidentally, VGA and other high-resolution graphics boards are availble for PC-, PC/XT-, and PC/AT-type computers.

In addition to built-in VGA capability, the PS/2 models 60, 70, and 80 have an I/O slot especially designed for a high-resolution graphics board such as the 8514/A. The 8514/A uses a custom two-chip set to provide graphics modes with up to 1024 × 768 pixels and 256 from a palette of 256K.

Obviously we can't describe here all the details of all the graphics adapters and modes shown in Figure 13-14. In the following sections we will briefly discuss the hardware used to implement a CGA adapter, an EGA adapter, and a VGA adapter. In a later section we show you how to write characters or dots of a desired color to each of these basic display types.

THE IBM PC COLOR GRAPHICS ADAPTER BOARD

Figure 13-15, page 448, shows a block diagram of the IBM PC Color Graphics Adapter (CGA) board. This board again uses the Motorola MC6845 CRT controller device to do the overall display control. As we described in a

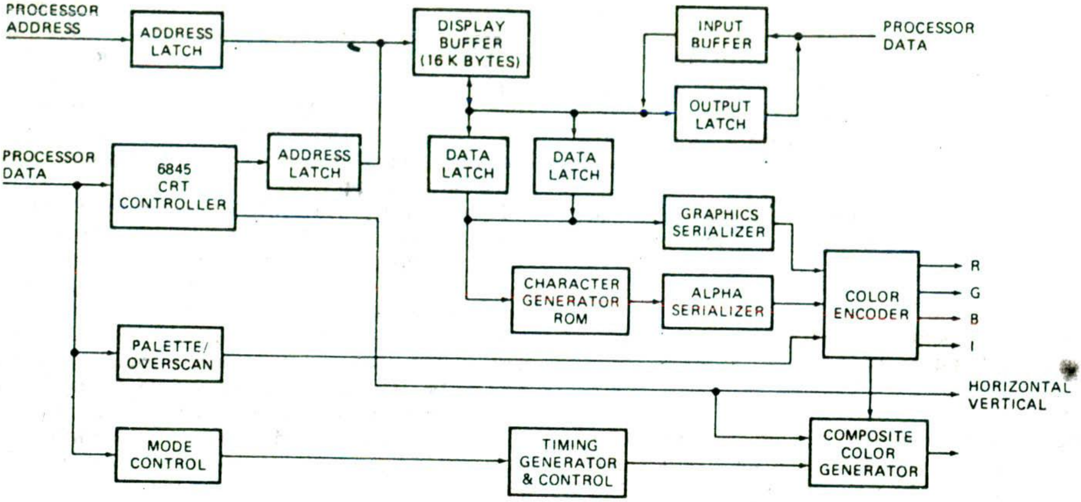


FIGURE 13-15 The block diagram of the IBM PC Color Graphics Adapter (CGA) board.

previous section, the 6845 produces the sequential addresses required for the display-refresh RAM, the horizontal sync pulses, and the vertical sync pulses. As you can see by the signals shown in the lower right corner of Figure 13-15, the adapter board is designed to drive either a monitor with separate red, green, and blue inputs or a *composite video color monitor*, which has all the required signals combined on a single line. The 16-Kbyte display-refresh RAM on the CGA board is *dual-ported* so that it can be accessed by either the system processor or the CRT controller.

This adapter board can operate in either a character mode or a graphics mode. In the character mode it uses a character-generator ROM and shift registers (alpha serializer) to produce the serial dot information for the RGB outputs. In the character or alphanumeric mode each character is represented by 2 bytes in the display-refresh RAM in the format shown in Figure 13-9*a*. The even-addressed or lower byte contains the 8-bit ASCII code for the character to be displayed. The odd-addressed or upper byte contains an attribute code, as shown in Figure 13-9*b*. The lower 4 bits of this attribute byte use the codes shown in Figure 13-11 to specify the color of the displayed characters. Bits 4–7 of the attribute byte allow you to specify the background color from among the first eight choices shown in Figure 13-11. The B bit in the attribute byte allows you to specify that the character will blink. Only 4 Kbytes are needed to hold the character and attribute codes for an 80-character by 25-row display, so the codes for up to four pages can be present in the 16-Kbyte display RAM.

As a preview of video programming, perhaps you can see how you can display a character at a particular location on the screen of a CGA system by directly writing to the display RAM. The frame buffer in a CGA system starts at absolute address B8000H and 2 bytes are required to hold the character code and attribute for each character. You can just count up by 2 from B8000H to get the address which corresponds to a particular character position on the screen. You then use a MOV instruction to write the ASCII code for the character to that address and the attribute byte to the next higher address.

When operating in a color graphics mode, a CGA board uses separate shift registers (graphics serializer) to produce the dot information for each of the color guns and for the overall intensity. The pixel data for the graphics serializer comes directly from the display-refresh RAM.

For displaying graphics, a CGA adapter board can be operated in low-resolution mode, medium-resolution mode, or high-resolution mode. The low-resolution mode is not of much interest, because the display has only 100 rows of pels with 160 pels in each row. The high-resolution mode displays 200 rows with 640 pels in each row, but it can produce only monochrome graphics displays.

In the medium-resolution mode the display consists of 200 rows of pels with 320 pels in each row, or a total of 64,000 pels. The 16 Kbytes of display-refresh RAM corresponds to 16 Kbits × 8 or 128 Kbits. Dividing the number of pels into the number of bits available for storage tells you that in this mode there are only 2 bits per pel available to store color information. With 2 bits you can specify only one of four colors for each pel.

Figure 13-16*a* shows how the 2 bits for each pel are positioned in display-refresh RAM bytes. Figure 13-16*b* shows the codes used to specify the color desired for a pixel. The 2 bits for each pel specify whether that pel is to have the background color or a color from one of two color sets. Figure 13-16*c* shows the two available color sets. The desired color set can be selected by outputting a control byte through port 3D9H to the palette circuit shown on the left edge of Figure 13-15. As we show you later in the section on video programming, an easier way to do it is with the BIOS INT 10H procedure.

7	6	5	4	3	2	1	0
C1	C0	C1	C0	C1	C0	C1	C0
FIRST DISPLAY PEL		SECOND DISPLAY PEL		THIRD DISPLAY PEL		FOURTH DISPLAY PEL	

(*a*)

C1	C0	FUNCTION
0	0	DOT TAKES ON THE COLOR OF 1 of 16 PRESELECTED BACKGROUND COLORS
0	1	SELECTS FIRST COLOR OF PRESELECTED COLOR SET 1 OR COLOR SET 2
1	0	SELECTS SECOND COLOR OF PRESELECTED COLOR SET 1 OR COLOR SET 2
1	1	SELECTS THIRD COLOR OF PRESELECTED COLOR SET 1 OR COLOR SET 2

(*b*)

COLOR SET 1	COLOR SET 2
COLOR 1 IS GREEN COLOR 2 IS RED COLOR 3 IS BROWN	COLOR 1 IS CYAN COLOR 2 IS MAGENTA COLOR 3 IS WHITE

(*c*)

FIGURE 13-16 CGA 320 × 200 pixel storage formats. (*a*) Position of pel bits in memory byte. (*b*) Codes used to specify the color desired for a pixel. (*c*) Two-color sets.

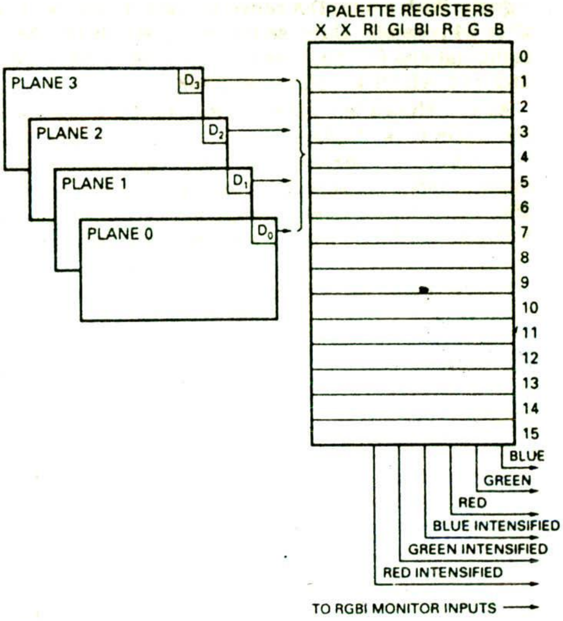


FIGURE 13-17 Functions of the frame buffer and palette registers for the most commonly used EGA display modes.

You can write dots directly to a CGA screen by writing the byte for each 4 pixels on a line to the appropriate memory location. A CGA adapter uses interlaced scanning, so the pixel codes for the even scan lines are put in memory starting at B8000H and the pixel codes for the odd scan lines are put in memory starting at BA000H. You can count up from these starting addresses to find the memory location that corresponds to a given pixel location on the screen. In a later section we show you how to use the BIOS INT 10H procedure to write dots to the screen.

ENHANCED GRAPHICS ADAPTER (EGA) HARDWARE

The Enhanced Graphics Adapter has a programmable CRT controller which allows you to set the display mode much as you do with the MC6845 on a CGA board. However, the key parts of EGA hardware that we need to talk about are the frame buffer and the palette registers. We can't show you how all the different EGA display modes use these, but Figure 13-17 shows how they function for the most commonly used format, a 640 × 350 pixel display with 16 colors from a palette of 64.

In this mode the frame buffer memory is configured as four planes. Each plane holds one of the 4 bits required to specify the color of each pixel. A 4-bit value read from the four planes is used to address one of the sixteen 8-bit palette registers. The lowest 6 bits from the addressed palette register are output to the color monitor.

NOTE: To work with this mode the monitor must have red-intensified, green-intensified, and blue-intensified inputs as well as standard red, green, and blue inputs.

There are 64 possible combinations for the 6-bit value in each palette register, but since there are only 16 palette registers, only 16 of the 64 possible combinations can be stored at a time. The 16 values in the palette registers at any particular time then specify 16 colors from a palette of 64. During bootup the palette registers in an EGA are initialized with values which correspond to the 16 colors available on a CGA system, but you can load the palette registers with values which produce your favorite colors.

VIDEO GATE ARRAY (VGA) DISPLAY HARDWARE

A proprietary gate array CRT controller device in a VGA based system allows you to select the dot clock frequency, the number of horizontal scan lines, the vertical refresh rate, the amount of overscan, etc. You can program a VGA system to operate in the previously described CGA modes, in the various EGA modes, or in several other modes. One of the standard display modes of a VGA allows you to display up to 256 colors at a time from a palette of 262,144 colors. In order to produce 256 colors, a VGA system generates analog red, green, and blue signals instead of the digital RGB signals used by previously described CRT adapters. As shown in Figure 13-18*a* (page 450), a 6-bit D/A converter, commonly called a *video DAC*, is used to produce each of the color signals. With a 6-bit D/A converter each signal can have 2^6 or 64 possible values, so the total possible number of combinations for the three signals is 64 × 64 × 64 or 262,144, which we refer to as 256K.

As also shown in Figure 13-18*a*, the 18-bit values for the colors to be displayed are stored in 256 color registers. Since there are only 256 color registers, the maximum number of colors that you can display at a time is 256. Incidentally, several companies produce ICs called *RAMDACs*, which contain both the color registers and the D/A converters.

For each pixel an 8-bit value is used to select the color register which drives the D/A converters. This 8-bit value is produced in several different ways, depending on the selected display mode.

The left side of Figure 13-18*a* shows how this 8-bit color register "address" is generated for a 320 × 200 × 256 color display. The 8-bit values for four successive pixels are stored in four memory planes as shown. When an 8-bit pixel value is read from one of the memory planes, the upper 4 bits of the pixel value are used directly as part of the color register address. The lower 4 bits of the pixel value from the memory plane are used to address one of 16 palette registers. (These palette

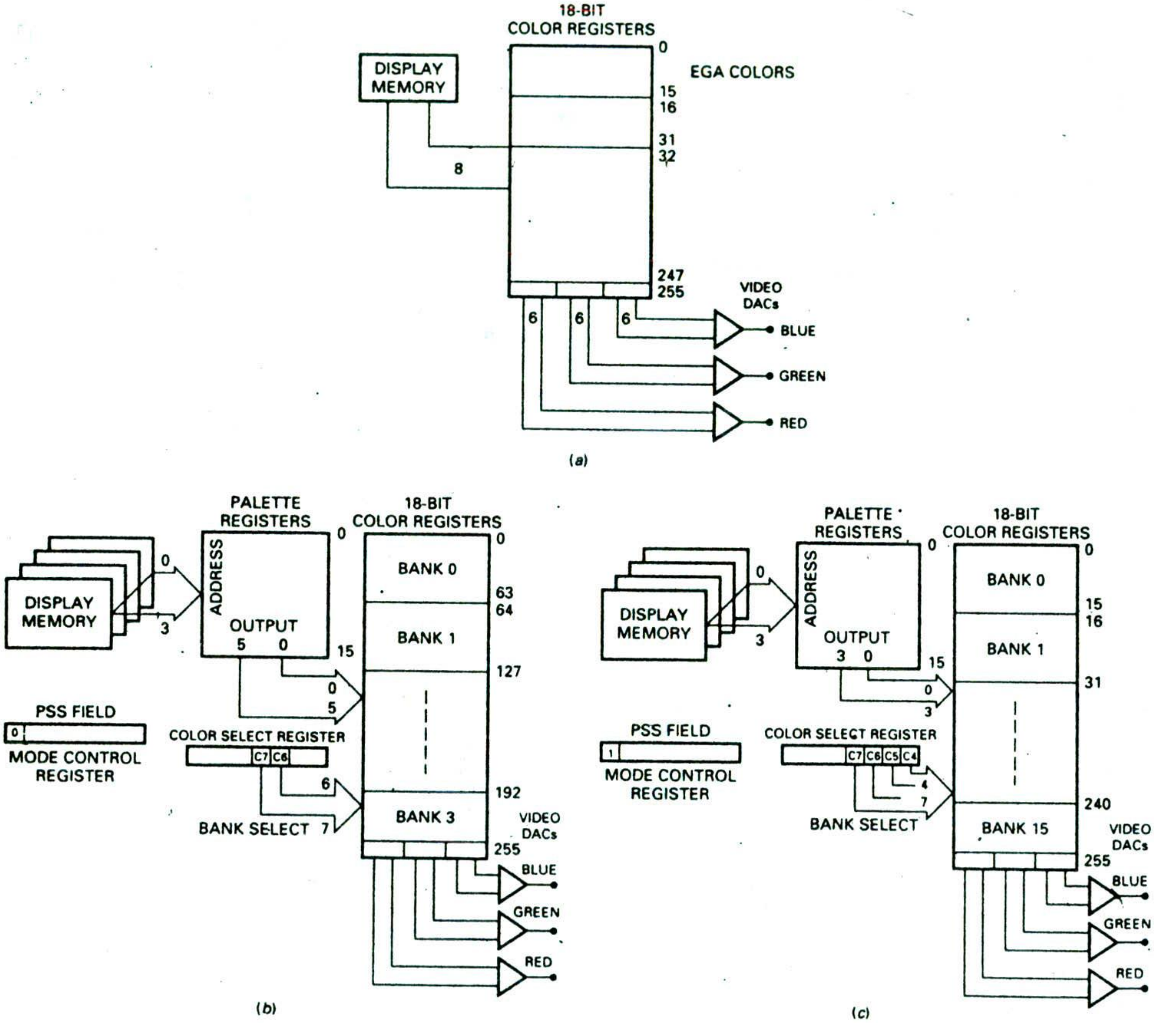


FIGURE 13-18 Hardware configurations for common VGA display modes. (*a*) 320 × 200 × 256 color—8 bits per pixel directly select one of 256 color registers. (*b*) 640 × 480 × 16 color—2 bits from color-select register select one of four banks of 64 color registers. Four bits from color planes select one of 16 palette registers. Six-bit value from palette register selects one of the 18-bit color registers. (c) Alternative 640 × 480 × 16 color—4 bits from color-select register select a bank of 16 color registers. Lower 4 bits from palette register value select one of 16 color registers in that bank.

registers were included for downward compatibility with EGA graphics modes.) The lower 4 bits of the value from the addressed palette register are used as the lower 4 bits of the color register address.

Figure 13-18*b* shows one way the VGA hardware is configured for a 640 × 480 pixel display with 16 colors. The 4-bit pixel values are stored in four bit-mapped planes. To the CRT controller each of these bit planes has the same address, so the controller reads out all the bits for a pixel at the same time. The 4 bits read from the four memory planes are used to address one of the 16 palette registers. The lower 6 bits from the addressed palette register are used as the lower 6 bits of the 8-bit address for the color registers. A 6-bit value from the palette register can address any one of 64 color registers. However, since there are only 16 palette registers, only 16 of the 64 color registers can be accessed at a time. This means that the display can only have 16 colors at a time.

The upper 2 bits of the address for the color registers come from bits C7 and C6 of a special register called the Color Select Register. These bits are user programmable. You can think of these bits as selecting one of four banks of 64 colors in the 256 color registers. You can change these bits to rapidly change from one set of 16 colors to another.

Figure 13-18c shows another way the VGA hardware can be configured for a 640 × 480 pixel display with 16 colors. In this configuration the 4 bits from the memory planes select one of the 16 palette registers, but only the lower 4 bits of the 6 bits from the palette register are used to address the color registers. The other four bits of the color register address come from the Color Select Register. You can think of these four bits as selecting one of 16 banks of colors in the 256 color registers. Again, the advantage of this approach is that you can very quickly switch from one set of 16 colors to another.

After reading through the preceding discussions of the CGA, EGA, and VGA adapters the question that probably occurs to you is, With all these different pixel storage formats, palette registers, and color registers, how on earth do I put text or graphics to the screen as part of a program? In the next section of the chapter we show you some ways to do this.

Video Programming

INTRODUCTION

There are three major methods of writing programs to control the video hardware we have described in the preceding sections. One method is to use assembly language instructions to write the required values directly to the CRT controller registers, the palette registers, the frame buffer, etc. A program written at this level usually runs faster than one written at a higher level, but programming at this level is quite complex because you have to keep track of many register bits, memory addresses, pixel storage formats, etc. Also, when programming at this register level it is easy to forget to restore some register to the correct value at the end of an operation. This may cause the display to "hang."

The next-higher level of CRT programming is to use the BIOS INT 10H procedures. In some cases the INT 10H procedures execute very slowly, but they are easy to use and programs written using them have a high degree of portability to different systems.

Still another level of video programming is to use a high-level language such as C. The advantage of this high-level-language approach, of course, is that you do not have to "reinvent the wheel." You can simply call library functions to draw lines and boxes, create windows, move images on the screen, etc. The Turbo C graphics functions manipulate the display hardware directly rather than using the BIOS INT 10H procedure, so they are quite fast. Our choice is to use C wherever possible, but your choice will depend on the programming environment you have available. To give you choices we will first show you how to use the BIOS INT 10H procedures to initialize a graphics adapter, write characters to the screen, and write pixels of a desired color to the screen. Then we will show you how to write pixel data directly to the frame buffer of an EGA or VGA system. Finally, we will show you how to use some of the Turbo C graphics functions.

USING BIOS INT 10H PROCEDURE FOR VIDEO PROGRAMMING

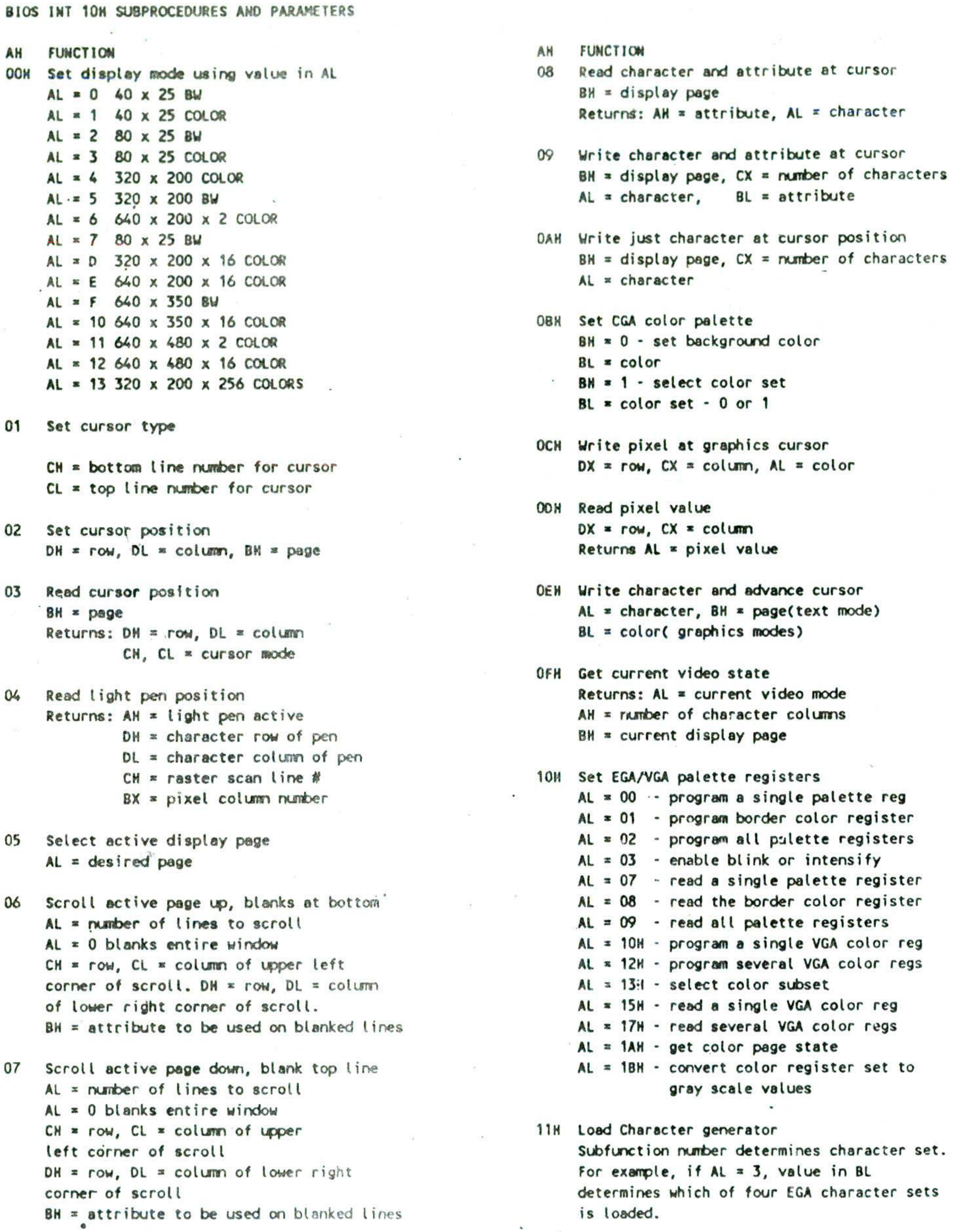
Figure 13-19, page 452, shows you the different subprocedures or functions of the BIOS INT 10H procedure and the registers used to pass parameters to these functions. Some of these functions are: set display mode, set cursor position, scroll page up, scroll page down, write dot, and write character to screen. An important point here is that the BIOS procedures on an EGA system are a "superset" of the BIOS procedures for a CGA system. The INT 10H procedure in an EGA system simply has additional subprocedures to access the palette registers, etc. on an EGA system. Likewise, the VGA BIOS INT 10H procedures are a superset of the BIOS procedures for CGA and EGA systems. The significance of this is that a VGA system, for example, can be programmed to operate as an EGA system or as a CGA system so that programs written for those systems can be run without problems.

The first step in a video program is to set the system to the desired display mode. An INT 10H procedure call is an easy way to do this. As a simple example of this, the following instructions use the BIOS INT 10H to initialize a CGA, EGA, or VGA adapter to a color text mode with 80 columns × 25 rows.

```
MOV AH, 00    ; Call set mode function of INT 10H
MOV AL, 03    ; Code for 80 × 25 color text mode
INT 10H       ; Call BIOS procedure
```

To put a system in some other mode you simpy put the display mode number from Figure 13-19 in AL instead of the 03H used for this example.

Once you get the adapter or system in the desired display mode, the next step is to write characters or dots to the desired positions on the screen. The following instructions use the BIOS INT 10H procedure to put the



```
BIOS INT 10H SUBPROCEDURES AND PARAMETERS

AH   FUNCTION
00H  Set display mode using value in AL
     AL = 0  40 x 25 BW
     AL = 1  40 x 25 COLOR
     AL = 2  80 x 25 BW
     AL = 3  80 x 25 COLOR
     AL = 4  320 x 200 COLOR
     AL = 5  320 x 200 BW
     AL = 6  640 x 200 x 2 COLOR
     AL = 7  80 x 25 BW
     AL = D  320 x 200 x 16 COLOR
     AL = E  640 x 200 x 16 COLOR
     AL = F  640 x 350 BW
     AL = 10 640 x 350 x 16 COLOR
     AL = 11 640 x 480 x 2 COLOR
     AL = 12 640 x 480 x 16 COLOR
     AL = 13 320 x 200 x 256 COLORS

01   Set cursor type

     CH = bottom line number for cursor
     CL = top line number for cursor

02   Set cursor position
     DH = row, DL = column, BH = page

03   Read cursor position
     BH = page
     Returns: DH = row, DL = column
              CH, CL = cursor mode

04   Read light pen position
     Returns: AH = light pen active
              DH = character row of pen
              DL = character column of pen
              CH = raster scan line #
              BX = pixel column number

05   Select active display page
     AL = desired page

06   Scroll active page up, blanks at bottom
     AL = number of lines to scroll
     AL = 0 blanks entire window
     CH = row, CL = column of upper left
     corner of scroll. DH = row, DL = column
     of lower right corner of scroll.
     BH = attribute to be used on blanked lines

07   Scroll active page down, blank top line
     AL = number of lines to scroll
     AL = 0 blanks entire window
     CH = row, CL = column of upper
     left corner of scroll
     DH = row, DL = column of lower right
     corner of scroll
     BH = attribute to be used on blanked lines
```

```
AH   FUNCTION
08   Read character and attribute at cursor
     BH = display page
     Returns: AH = attribute, AL = character

09   Write character and attribute at cursor
     BH = display page, CX = number of characters
     AL = character,    BL = attribute

0AH  Write just character at cursor position
     BH = display page, CX = number of characters
     AL = character

0BH  Set CGA color palette
     BH = 0 - set background color
     BL = color
     BH = 1 - select color set
     BL = color set - 0 or 1

0CH  Write pixel at graphics cursor
     DX = row, CX = column, AL = color

0DH  Read pixel value
     DX = row, CX = column
     Returns AL = pixel value

0EH  Write character and advance cursor
     AL = character, BH = page(text mode)
     BL = color( graphics modes)

0FH  Get current video state
     Returns: AL = current video mode
     AH = number of character columns
     BH = current display page

10H  Set EGA/VGA palette registers
     AL = 00  - program a single palette reg
     AL = 01  - program border color register
     AL = 02  - program all palette registers
     AL = 03  - enable blink or intensify
     AL = 07  - read a single palette register
     AL = 08  - read the border color register
     AL = 09  - read all palette registers
     AL = 10H - program a single VGA color reg
     AL = 12H - program several VGA color regs
     AL = 13H - select color subset
     AL = 15H - read a single VGA color reg
     AL = 17H - read several VGA color regs
     AL = 1AH - get color page state
     AL = 1BH - convert color register set to
                gray scale values

11H  Load Character generator
     Subfunction number determines character set.
     For example, if AL = 3, value in BL
     determines which of four EGA character sets
     is loaded.
```

FIGURE 13-19 BIOS INT 10H subprocedures and parameters.

cursor at position 24 in row 7 and write a blinking A to the screen at that position.

```
MOV AH,02H   ; Load subfunction number for set cur-
                sor position
MOV DH,07    ; load row number of cursor position
MOV DL,24    ; load column number of cursor position
MOV BH,00    ; load display page number
INT 10H      ; call BIOS video procedure
             ; write character at cursor position
MOV AH,09    ; subfunction # for write character/attri-
                bute at cursor position
MOV BH,00    ; load display page number
MOV CX,01    ; load number of characters to send
MOV AL,41H   ; ASCII code for A
MOV BL,      ; Attribute code for blinking,
  10001001B  ; black background, light blue character
             ; See Figure 13-9 for attribute byte format
INT 10H      ; Call BIOS video procedure
```

The first call of the INT 10H puts the cursor in the desired location on the screen. Note that as part of the setup for this call, we loaded the desired display page in the BH register. As we pointed out earlier, an 80 × 25 character display requires only 4 Kbytes of frame buffer memory. The IBM CGA board has a 16-Kbyte frame buffer, so with one of these boards you can have up to four pages stored in the frame buffer at a time. You can use subfunction 5 of the INT 10H procedure to flip the display from one page to another.

The second call of the INT 10H procedure simply sends the character and attribute bytes to the correct locations in the frame buffer. The character code to be sent is put in AL and the attribute byte is put in BL. If CX contains a 1, the character will be written to just the location where the cursor is located. After the write the cursor will not be advanced to the next position. If CX contains a number other than 1, the same character will be written to the number of sequential locations contained in CX. The cursor will be left on the last character written.

If you want to write a sequence of characters to the display, function 14 of the INT 10H procedure is more efficient because it automatically advances the cursor to the next position after a write. To call this function you load 14 in AH, load the ASCII code for the character to be sent in AL, and execute the INT 10H instruction.

The obvious advantage of using the BIOS procedure here is that you don't have to figure out the memory location which corresponds to the position of the character on the screen.

Incidentally, a CGA system has only one text font and the character generator for that font is in ROM, as shown in Figure 13-15. On an EGA or VGA system the character generator is located in RAM. The default character generator data for a specified mode is loaded into RAM when the system is put in that mode with an INT 10H function. However, you can use subfunction 17 of the INT 10H procedure to load one of several available character fonts or a custom character font.

Now that you know how to display characters on the screen, let's take a look at how you can produce a graphics display. Earlier we mentioned that you could write a dot on the screen by writing the appropriate pixel code directly to the corresponding memory location. An easier way to do this is with the BIOS INT 10H procedure.

The program in Figure 13-20, p. 454, shows you how to put an EGA or VGA system in 640 × 350 × 16 color mode, set the background color for light blue, draw a magenta window near the center of the screen, and write a message in the window. The first major step in the program is to use the INT 10H procedure to put the system in mode 10H, which according to Figure 13-19 is the 640 × 350 × 16 color graphics mode. The next step in the program is to set the background color to light blue. For an EGA or VGA in this mode, the background color is the color code stored in palette register zero. To change the background color, then, all you have to do is use subfunction 16 of the INT 10H procedure to load the code for the desired color in palette register 0.

The next part of the program uses a nested loop and the "write dot" subfunction of INT 10H to draw a magenta window on the screen. The window is produced by drawing horizontal lines. You can draw a window of any size by simply changing the start and stop coordinates in this loop.

The final section of the program in Figure 13-20 uses function 2 of INT 10H to position the cursor in the window and then uses function 14 to write a message at the cursor position. We included this last section to show you that you can use the INT 10H text functions to write characters to the screen even though you are in graphics mode. After writing the message to the screen we load AX with 4C00H and use software interrupt 21H to return execution to the DOS prompt.

A DIRECT WRITE VIDEO GRAPHICS EXAMPLE

The program in Figure 13-20 takes about 10 s just to draw the small window on the screen of a 25-MHz 80386-based microcomputer. For some applications this may be an unreasonably long time, so we decided to show you how to do the job by directly manipulating the controller registers and writing to the video RAM.

The program in Figure 13-21, p. 455, puts an EGA or VGA system in the 640 × 350 × 16 color graphics mode and draws a magenta window just as the program in Figure 13-20 does, but to save paper we did not show the write message portion. This program draws the window in less than 1 s. The program would be much faster, except for the fact that when an EGA or VGA system is operating in a high-resolution graphics mode, the processor can access the video RAM only about 20 percent of the time.

After we initialize the system to the desired mode and set the background color with the INT 10H procedure, we use three custom procedures to set the controller registers, write dots to draw the window, and restore the controller registers to their initial values. We don't have space here to discuss all the details of the EGA and VGA controller registers, but we will try to give you

```
; 8086 PROGRAM F13-20.ASM
;ABSTRACT : This program use the BIOS INT 10H procedure to put
          ; an EGA or VGA system in 640 x 350 x 16 color graphics
          ; mode, set the background to light blue, draw a magenta
          ; window and display a message in the window

DATA SEGMENT
     TEXT  DB 'GRAPHICS PROGRAMMING IS FUN',24h
DATA ENDS

CODE SEGMENT
     ASSUME CS:CODE, DS:DATA
START: MOV AX, DATA             ; Initialize DS
       MOV DS, AX
       MOV AL, 10H              ; Set up for 640 x 350
       MOV AH, 0                ;    graphics mode
       INT 10H
       MOV AH, 10H              ; Set background color light blue
       MOV AL, 0                ;  with subprocedure 10H of BIOS
       MOV BL, 0                ;  INT 10H. AL = palette function
       MOV BH, 09               ;  BL = reg #, BH = color
       INT 10H
       MOV  BX, 0005H           ; Display page and color for window
       MOV  CX, 160             ; start column number for window
       MOV  DX, 100             ; start row number for window
L1:    MOV  AH, 0CH             ; INT 10H write dot sub function
       MOV  AL, BL              ; color from store
       INT  10H                 ; video BIOS routine
       INC  CX                  ; Increment column count
       CMP  CX, 480             ; check for end of row
       JB   L1                  ; No, write another dot
       INC  DX                  ; Yes, increment row count
       CMP  DX, 250             ; Check if all rows done
       JE   DONE                ; Yes, go write message
       MOV  CX, 160             ; No, point at start of line
       JMP  L1                  ; Draw next dot row
DONE:  MOV AH, 02               ; Set cursor position
       MOV DH, 12               ; Load character row number
       MOV DL, 27               ; Load character column number
       MOV BL, 0                ; Display page number
       INT 10H
; Write messsage in window
       MOV DX, 00               ; Use DX to hold pointer
       MOV BL,0FH               ; BL contains desired character color
NXTCHR:MOV SI, DX               ;   SI destroyed in INT 10H
       MOV AH, 14               ; Write character at cursor and
       MOV AL, TEXT[SI]         ;   increment cursor position
       CMP AL, 24H              ; Check if sentinel character
       JE EXIT
       INT 10H
       INC DX                   ; Point to next character in string
       JMP NXTCHR
EXIT:  MOV AX, 4C00H            ; return to DOS
       INT 21H
CODE   ENDS
       END START
```

FIGURE 13-20 Program using BIOS INT 10H to draw a window on an EGA or VGA.

enough information so you can comfortably use parts of Figure 13-21 in your own programs. For further details consult one of the EGA/VGA programming books listed in the Bibliography.

As the name implies, the SETUP procedure gets the attention of the video controller device and puts some of its registers in the required mode. EGA and VGA controllers each have an incredible number of registers

```
; 8086 PROGRAM F13-21.ASM
; ABSTRACT: This program puts an EGA or VGA in 640 x 350 x 16 color
          ; graphics mode, sets the background light blue, then
          ; uses direct controller writes to draw a magenta window.

CODE     SEGMENT
     ASSUME CS:CODE
START:
     MOV AL, 10H               ; Put system in 640 x 350
     MOV AH, 0                 ; graphics mode
     INT 10H                   ;
     MOV AH, 10H               ; INT 10 H subprocedure #
     MOV AL, 0                 ; Write to single palette register
     MOV BL, 0                 ; Palette register number
     MOV BH, 09H               ; Code for light blue
     INT 10H
     CALL SETUP                ; Set up graphics controller for set/reset
                               ; mode of writing to display buffer
     MOV  BX, 160              ; start column number for window
     MOV  AX, 100              ; start row number for window
L1:  CALL WRITE_DOT            ; Fast write pixel procedure
     INC  BX
     CMP  BX, 480              ; check for end of row
     JB   L1
     INC  AX                   ; Increment row count
     CMP  AX, 250              ; Check if all rows done
     JE   EXIT                 ; Yes, done
     MOV  BX, 160              ; No, point at start of line
     JMP  L1                   ;   Draw next dot row
EXIT:CALL RESTORE
     MOV AX, 4C00H             ; return to DOS
     INT 21H

PROC SETUP
     MOV DX, 03CEH             ; Address of controller reg
     MOV AX, 0005h             ; Enable Write Mode 0. AH-write mode, AL-index
     OUT DX, AX
     MOV AH, 05                ; Load Set/Reset register with
     MOV AL, 0                 ; code for magenta,  AL - index
     OUT DX, AX
     MOV AX, 0F01h             ; Enable all four color planes
     OUT DX,AX                 ;  AH - enable bits, AL - index
     RET
ENDP

PROC WRITE_DOT
; The Video Controller must be set for Write Mode 0, and have the
; Map Mask Set for the desired color outputs.
     PUSH AX
     PUSH BX
     PUSH CX
     PUSH DX
; Compute address of pixel in video buffer. Pixel address = row x 80 + column/8
     MOV DX, 80
     MUL DX                    ; AX now = row * 80
     MOV CX, BX                ; Save column for later use
     SHR BX, 1
     SHR BX, 1
     SHR BX, 1                 ; BX = col / 8
     ADD BX, AX                ; BX = row * 80 + col / 8
     MOV AX, 0A000h            ; Point ES at video buffer base
     MOV ES, AX
```

FIGURE 13-21 Program using direct controller write to draw a window.

```
        AND CL, 7                     ; Use lowest 3 bits in column
                                      ; to determine pixel # in byte
        MOV AX, 8008h                 ;  generate and set the BitMask
        SHR AH, CL                    ;   in graphics controller, so
        MOV DX, 03CEh                 ;   so. don't write color to all
        OUT DX, AX                    ;   8 pixels in byte
        OR ES:BX, AL                  ; Write the Pixel - (Contents of AL ignored).
        POP DX
        POP CX
        POP BX
        POP AX
        RET
ENDP

PROC RESTORE
        MOV DX, 03CEH
        MOV AX, 0000H                 ; Default Set/Reset register value
        OUT DX, AX
        MOV AX, 0001H                 ; Default Enable Set/Reset value
        OUT DX, AX
        MOV AX, 0FF08H                ; Default bit mask value
        OUT DX, AX
        RET
ENDP
CODE ENDS
        END START
```

FIGURE 13-21 *(Continued)*

that are used to hold the values of parameters for various display modes, etc. To reduce the number of I/O addresses required to access all these registers, an index system is used. Here's how it works.

Each group of registers in the controller has two I/O addresses. As an example, the group of registers we access in this program uses the addresses 03CEH and 03CFH. The lower address is used to send the index number for the register we want to access, and the upper address is used to write the desired value to the selected register or read a value from the selected register. To speed up write operations, the index and the value can be sent with a single 16-bit OUT instruction.

There are several ways to write pixel values to the memory planes of an EGA or VGA system. For this example we chose the "set/reset" method, because it is very efficient for drawing lines or filling regions of the screen. To give you an overview before we get into the details, the major steps in this method are

1. Put the video controller in write mode 0 so that the set/reset write mode will work.
2. Enable the planes we want affected by the color value we will load into the set/reset register.
3. Load set/reset register with the desired color value.
4. Generate and send a mask byte so that only the desired pixel bits in the display memory bytes are set or reset.
5. Activate the controller to set/reset the desired bits in the display memory.

The first group of instructions in the SETUP procedure enables the controller for Write Mode 0 so the set/reset operation will work. The second MOV and OUT group of instructions in the SETUP procedure is used to enable the desired color planes in the video RAM for a write. In this case we want to write to all four planes, so we put 1's in the lower 4 bits of the data word in AH. The SETUP procedure needs to be done only once before a series of dots is written.

The next procedure to look at in Figure 13-21 is the WRITE_DOT procedure. The first task of the WRITE_DOT procedure is to compute the video RAM address which corresponds to the desired pixel coordinates. As shown in Figure 13-17, the pixel data for this display mode is stored in four parallel planes. The 4-bit value for each pixel is stored as the same numbered bits in the four planes. Each byte in one of the planes then contains 1 bit of the pixel data for 8 pixels, so it takes 80 bytes of memory in each plane to store the pixel data for one line of 640 dots on the screen. From the microprocessor's standpoint the four planes all occupy the same system address space, starting at address 0A000H. The offset of the byte which contains the bit for a particular pixel then can be calculated with the simple expression Offset = (row × 80) + column/8. In the WRITE_DOT procedure in Figure 13-21, we used the MUL instruction to multiply the row number by 80, but for the divide instruction we used a shift-right operation because it is faster than DIV.

The next step in the WRITE_DOT procedure is to generate a mask which will be used to make sure a new pixel code is written only to the desired bit in each of the display planes. The number of a bit in a byte is represented by the lowest three bits of the column number. The MOV CX,BX instruction in the procedure saves the column number in CX, so the AND CL,07H instruction gives the number of the bit in the memory byte. To actually generate the mask word, we load 80H in AH and shift this value CL times to the right. The 1 in the data word in AH will be left in the bit position we want to write to in the RAM byte. (As we show you in a problem at the end of the chapter, it is sometimes useful to set all the bits in this mask so that the pixel data is written to all 8 bits in a byte at the same time.) Once the mask is generated we send it and the index of 08H in AL out to the controller.

As we mentioned before, the four planes of the video buffer RAM all occupy the same address space, starting at 0A000H. For the write mode we have chosen, this means that we have to send pixel codes to the RAM through the controller rather than writing them directly. Remember that in the WINDOW procedure, we sent the pixel code to the Set/Reset (color) register of the controller. The first step in this indirect process is to point ES at the base of the Video RAM segment. The next step is to tell the controller to write the pixel code to the desired address with the OR ES:[BX],AL instruction.

Normally, the OR memory, register instruction reads a byte from memory, ORs AL with this word, and writes the result back to the specified memory location. In this program the read part of the operation causes the controller to read a byte from each of the display planes into four latches in the controller. The controller then sets or resets the unmasked bits in each of these registers with the new pixel code. During the write part of the OR operation the updated bytes are written back to the specified address in the planes. The byte in AL is ignored during this operation.

The RESTORE procedure at the end of the program is used to return all the registers in the video controller to their default values. We wanted to show you how to do this, but in most cases it is not really necessary, because other procedures will usually put the controller in the mode needed for that procedure.

DRAWING DIAGONAL LINES ON THE SCREEN

The program in Figure 13-21 uses the WRITE_DOT procedure to draw horizontal lines across the screen. Perhaps you can see that the WRITE_DOT procedure could easily be used to draw a vertical line on the screen. However, drawing a diagonal line is much more difficult, because most of the pixel positions do not fall exactly on the desired line. As an example of this, Figure 13-22c shows the pixels which best approximate a line from coordinates (0, 0) to (10, 4). For each horizontal pixel position a calculation must be done to determine which of two vertical positions more closely approximates the desired line. To compute the "best-fit" pixel locations, we usually use Bresenham's algorithm. This algorithm determines the closer pixel by determining whether a point halfway between the pixels is above or below the actual line. A couple of examples should help you see how this works.

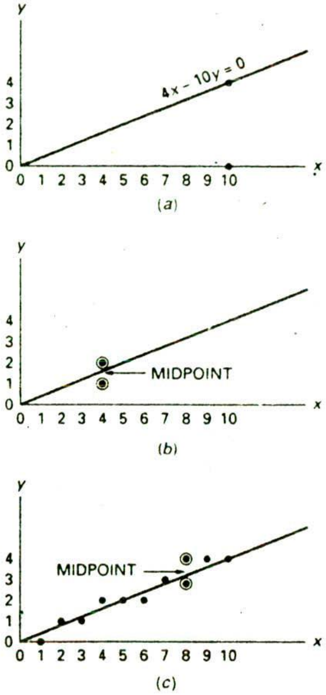


FIGURE 13-22 Drawing diagonal lines with pixels. (*a*) Graph of 4X × 10Y = 0. (*b*) Method to determine pixel to use for an X value of 4. (*c*) Method to determine pixel for X = 8 and best-fit pixels for entire line.

The equation for the line in Figure 13-22*a* is 4X − 10Y = 0. Figure 13-22*b* shows how we determine which pixel to use for an X value of 4. The two choices for the pixel are the Y = 1 pixel and the Y = 2 pixel. To determine which pixel, we first compute the value of the function at the midpoint between these two. For this example the midpoint is at (4, 1.5), so the function is 4 × 4 − 10 × 1.5 = +1. A positive result here indicates that the midpoint value is not large enough to make the function equal to zero; in other words, the midpoint is below the line. This means that the pixel at (4, 2) is closer to the actual line. In this case, all we do is increment the X value from that for the previous pixel, increment the Y value from that for the previous pixel, and write the pixel to the video RAM.

Figure 13-22*c* shows the results this technique produces for the pixel at X = 8. The midpoint here is (8, 3.5), so the function at this point is 4 × 8 − 10 × 3.5 = −3. The negative result tells you that the midpoint is above the line. This means that the pixel at (8, 3) is closer to the line than the pixel at (8, 4). In this case we increment the X value from that of the preceding pixel, keep the Y value the same as that of the preceding pixel, and write the pixel to the video RAM.

The example in Figure 13-22 assumes the line is in the first octant of a standard graph, but the basic principle can be extended to a line in any octant. To draw a line in the second octant, you increment the Y value by one and compute appropriate pixel value for X instead of incrementing X and computing Y. To draw a line in the third octant, you reverse the starting and ending points so the line is drawn left to right instead of right to left. We don't have space here to show you an assembly language program for this algorithm, but almost every graphics book has at least one.

One point we hope you gained from the preceding discussion is that it takes considerable computing just to draw a sequence of lines on the screen. Another type of graphics programming that requires considerable computation is the creation and manipulation of windows. The programs in Figures 13-20 and 13-21 showed you how to draw simple graphics windows by writing pixel data to locations in the frame buffer, but in most real applications there is considerably more to do than just draw the window.

Suppose, for example, that in a program you are writing you want to have a pull-down menu window which lists a series of commands that a user can select from, similar to the menus in the tc integrated environment. When you create a window, you write over the old pixel values in the frame buffer. Therefore, you must save the pixel codes for the region where you are going to put the window so that you can restore the original display when you close the window. The transfer of pixel codes from the frame buffer to another buffer, from the buffer back to the frame buffer, or from one location to another in the frame buffer is commonly called a bit-aligned block transfer or BITBLT (pronounced bit blit).

Diagonal lines and BITBLT operations are required in almost every graphics program, so predefined C functions have been developed to implement these and other graphics functions. In the next section we show you how to use some of the Turbo C predefined graphics functions.

USING C GRAPHICS FUNCTIONS

Almost every C compiler comes with a library of graphics functions. Also, several other companies market packages of graphics functions which provide capabilities beyond those of the basic graphic functions that come with the compilers. One difficulty with all these is that the names, operations, and prototypes for these functions vary widely from company to company. For our discussions here we use just the graphics functions that come with Turbo C version 2.0. If you have some other compiler, you should be able to find similar functions in the libraries for it.

Figure 13-23 shows a C program which uses Turbo C library functions to initialize the graphics adapter, draw an eight-segment "pie" graph on the screen, open a graphics window, draw some figures in the window, and close the window on user command.

To get an overview of how the program works, read just the comments in Figure 13-23; then read the discussion here to get the details of how the described action is done. You should then be able to use these functions to produce some interesting displays.

After we declare some variables, the first action in main is to call the initgraph function to initialize the display adapter. The arguments you pass this function are the address of a variable containing the desired graphics driver, the address of a variable containing the desired mode, and a pointer to a string containing the path to the specified graphics driver (.BGI) file. For this example we used the predefined constant VGA for the graphics driver and the predefined constant VGAHI for the display mode. These values produce a 640 × 480 16-color display. On an EGA system you could use the constants EGA and EGAHI to initialize the adapter for a 640 × 350 16-color display. Note in the string containing the path to the graphics drivers you have to insert the \ twice to get the compiler to accept a \.

The setbkcolor function next in the program sets the background color to the predefined constant LIGHTBLUE. The choices here are BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, LIGHTGRAY, DARKGRAY, LIGHTBLUE, LIGHTGREEN, LIGHTCYAN, LIGHTRED, LIGHTMAGENTA, YELLOW, and WHITE.

The for loop next in the program draws eight different-colored pie slices on the screen. The pie is centered at pixel coordinates X = 320 and Y = 175 and has a radius of 100 pixels. The starting point for the first slice is the 3 o'clock position. The angles are incremented so that the other slices are added in a counterclockwise direction. The predefined colors in the previous paragraph are really an enumerated list, where BLACK has a value of 0, BLUE has a value of 1, GREEN has a value of 2, etc. The color value of i + 1 that is passed to the setfillstype function starts with blue, then goes to green, then cyan, etc. as the loop executes.

After we draw the pie slices, we change the drawing color to white, draw a couple of white circles around the pie, and move the current position of the graphics cursor to the upper left corner of the screen for future reference.

> NOTE: The graphics cursor is normally invisible, so you have to keep track of where it is or, perhaps, on the screen draw an arrow which points to the current position.

The demo_window function called next in the program contains some really important programming points. As we mentioned, before you draw a window on the screen you have to first save everything from the current display so you can restore it when you close the window. This is equivalent to pushing registers on the stack when you call a procedure in assembly language.

The first data you need to save is the current position of the graphics cursor. The getx and gety functions determine the coordinates of the cursor and store them in the local variables X and Y.

The next data you have to save is the pixel data for the section of the screen where you are going to draw the window. The first step in this process is to call the imagesize function to determine how many bytes of memory are needed to store the pixel data. The value returned by imagesize is then passed to the function malloc. Malloc (memory allocate) creates a buffer of the requested size in an area of memory called *the heap*.

```
/* C PROGRAM F13-23.C */
/* Program to demonstrate Turbo C graphics functions and windows */

#include<stdlib.h>
#include<stdio.h>
#include<graphics.h>

void main()
{
int driver = VGA, mode = VGAHI;             /* 640 x 480 x 16 color */
int i, start_angl = 0, end_angl = 45;
void demo_window(void);                     /* function declaration */

initgraph(&driver,&mode, "c:\\tc");         /* initialize adapter */
setbkcolor(LIGHTBLUE);                      /* set background color */

for(i=0; i<8; i++)                          /* draw 8 pie wedges */
{
setfillstyle(SOLID_FI'L, i+1);              /* change drawing color for each */
pieslice(320,175,start_angl,end_angl,100);  /* center, angls, radius */
start_angl = start_angl + 45;               /* increment angles */
end_angl = end_angl + 45;
}
setcolor(WHITE);                            /* change drawing color to white */
circle(320,175,101);                        /* draw circles around pie */
circle(320,175,102);

moveto(0,0);                                /* set cursor to home position */
demo_window();                              /* call window demo function */
exit(0);                                    /* return to DOS */
}

void demo_window(void)
{
int x,y,i;
unsigned window_size;
void * window_buffer;

x=getx(); y=gety();                         /* get and save current cursor position */
                                            /* determine size of image to store */
window_size = imagesize(160,100,480,250);   /* find # bytes for image */
window_buffer = malloc(window_size);        /* dynamically allocate memory*/
getimage(160,100,480,250,window_buffer);    /* store bit map under window */
setviewport(160,100,480,250,1);             /* create window */
setcolor(RED);                              /* change drawing color */
for(i=0; i<150; i++)                        /* paint window red */
line(0,i,319,i);
setcolor(WHITE);                            /* change drawing color */
circle(25,25,10);                           /* draw circle in window */
rectangle(160,75,320,150);                  /* draw rectangle to show clipping */
while(getchar()!='e')                       /* wait for e to exit */
   continue;
setviewport(0,0,639,349,1);                 /* restore to full screen */
putimage(160,100,window_buffer,0);          /* restore original display */
free(window_buffer);                        /* release allocated memory */
moveto(x,y);                                /* put cursor back in position
                                               where it was before call */
}
```

FIGURE 13-23 C program which uses Turbo C library functions to initialize the graphics adapter, draw an eight-segment pie graph on the screen, open a graphics window, draw some figures in the window, and close the window on user command.

The heap is above the normal data area and below the stack. If there is not memory left in the heap for the requested buffer, malloc returns an error code of 00. Since this is such a small program, we did not bother to check for an error condition, but in a larger program you should. If malloc succeeds, it returns a pointer to the start of the buffer. Once you have the buffer set up, you use the getimage function to copy the pixel data from the display refresh RAM to the buffer. The arguments passed to getimage are: the X and Y coordinates of the upper-left corner of the area, the X and Y coordinates of the lower-right corner of the area, and a pointer to the buffer created by malloc. Now you are ready to actually create the window.

The setviewport function is used to create the window. The first four arguments passed to setviewport are the coordinates of the upper-left and the lower-right corners of the window. The final argument in the setviewport call is the clip flag. If this flag is set and you attempt to draw a figure which would extend outside the window, the figure will be clipped so it stops at the edge of the window.

The setviewport function creates the window and moves the graphics cursor to the upper left corner of the window. However, the created window has the same background color as the main screen, so it is not visible. To make th window visible we set the drawing color to red and draw the window full of lines. There are other ways to fill a rectangle, but this method gave us an excuse to show you how to draw lines.

Just for fun we drew a small circle and a rectangle in the window. Note that the coordinates for the circle are relative to the upper left corner of the window, not the upper left corner of the screen as they were for the preceding circles. Also note that the coordinates given for the rectangle would cause it to extend outside the window. We set the clip flag in the setviewport call, so the rectangle will not be allowed to extend outside the window. This is important, because anything drawn outside the window would not be replaced by the original display when you close the window.

When the user presses the e key and then the Enter key, we close the window and restore everything. The setviewport call here sets up the whole screen as the window again. The putimage function copies the pixel data from the window_buffer back to the display refresh RAM. The free(window_buffer) function call releases the memory allocated for the window_buffer by malloc. This step is very important, because if the memory is not put back in the heap when you are done with it, the heap will grow up until it runs into the stack. The principle here is the same as cleaning up the stack when returning from a function call.

The final step before returning to main is to put the graphics cursor back in the location where it was before the call. The moveto function puts the graphics cursor at the X and Y coordinates passed to it.

The C library functions we used in this program are obviously much easier to work with than the assembly language equivalents. They write directly to the video hardware instead of using the INT 10H procedures, so they are quite fast. However, they are still relatively low level. More advanced graphics packages such as Windows from Microsoft, DGIS from Graphic Software Systems, and GEM from Digital Research contain all the basic procedures needed to create and move windows. Also, they contain tools needed to draw text and graphics in windows. Packages such as this better insulate an applications programmer from the hardware details.

HIGH-RESOLUTION GRAPHICS AND GRAPHICS PROCESSORS

In the preceding sections we mostly discussed CGA, EGA, and VGA systems so that you could experiment with the examples on your own system. For many applications such as advanced CAD (computer-aided drafting), CAE (computer-aided engineering), or EDA (electronic design automation), however, the resolution of even a VGA screen is not nearly enough. For these applications a screen resolution of 1024 × 768 or greater is required. There are several problems in producing these high-resolution displays.

First, it is nearly a full-time job keeping the display refreshed and the display RAM refreshed. Second, manipulating complex images on the screen requires a great many computations. As we discussed earlier, even drawing a diagonal line or a circle on the screen requires considerable computation to determine the pixel values. Drawing a three-dimensional view of an airplane model on the screen, for example, requires several hundred thousand floating-point computations. Rotating the image to see a slightly different view requires hundreds of thousands of floating-point computations. Manufacturers have attempted to solve these problems in a number of ways.

As shown in Figure 13-14, IBM's next step up from the VGA is the 8514/A adapter, which plugs into a special slot in PS/2-type computers. This adapter uses a two-chip graphics processor to produce a display of 1024 × 768 pixels. One part of the 8514/A chip set keeps the display and the frame buffer RAM refreshed. The graphic processor part has hard-wired instructions to draw lines, perform BITBLT operations, and generate addresses for pixel coordinates. The 8514/A graphics processor, however, does not have instructions for drawing arcs and other more complex shapes, so the burden for these operations is still left with the main processor.

Another common example of a "hard-wired" graphic processor is the Intel 82786 Graphics coprocessor, which works with the main processor in about the same way that 8087 math coprocessor we described in Chapter 11 works with the main processor. The 82786 contains circuitry to refresh DRAMs as well as circuitry to refresh the screen. The 82786 has hard-wired instructions to draw lines, draw polygons, perform BITBLT operations, create multiple windows, and manipulate windows. The advantages of these hard-wired instructions are that they execute very rapidly and they require very little main processor overhead. The disadvantage of the 82786 is that if the desired operation is not one of the hard-wired instructions, you have to implement it with main processor instructions.

Still other common examples of graphics processors

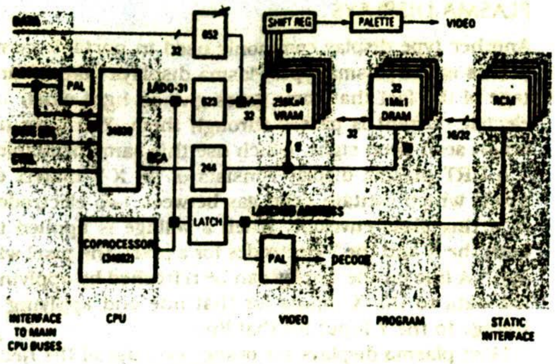


FIGURE 13-24 A TMS34020 ("three-forty-twenty") subsystem which can be connected to the main processor buses to control a 1024 × 1024 × 8-bit/pixel display.

are the Texas Instruments TMS34010 and TMS34020. Figure 13-24 shows how a TMS34020 ("three-forty-twenty") subsystem can be connected to the main processor buses to control a 1024 × 1024 × 8 bits/pixel display. As we explained earlier, the VRAM is used for the actual video buffer, because it can get data out fast enough to refresh pixels. The outputs of the VRAMs go to the palette registers, and the outputs of the palette registers go to D/A converters, which produce the analog RGB signals for the monitor. The standard DRAM and ROM in this subsystem are used to hold the programs and data for the graphics coprocessor. Note that a TMS34082 floating-point graphics coprocessor can be connected in the system to handle floating-point computations.

The major advantage and the major disadvantage of a TMS34020 system is its programmability. The device has basic graphics instructions such as draw line, pixel move, and area fill, and it can be programmed to implement much more complex graphics functions than common hard-wired devices, but the programmer has the burden of developing the required programs. To make this easier TI has a development system and an extensive library of programs for common operations.

Another step that Texas Instruments has taken to make the programmer's job easier is to develop the Texas Instruments Graphics Architecture (TIGA) standard and release this standard to the industry. TIGA establishes a standard interface between PC applications and the TMS34010 or TMS34020 hardware. What this means is that an applications programmer developing, for example, a drafting program can write his or her program to interface with TIGA and not worry about the actual hardware underneath it. TIGA does, however, allow the developer to write custom extensions if needed.

A still-higher level of graphics hardware is the Intel i860™, whose architectural block diagram is shown in Figure 13-25. This device is intended for use as the CPU in an engineering workstation which has advanced 3-dimensional imaging capability. As we discussed before, this requires a great many floating-point computations. With a 40-MHz clock the i860™ can perform at a peak rate of 80 *million floating point operations per second* (*MEGAFLOPS* or *MFLOPS*). The i860™ achieves this speed by putting many functions on a single chip and by doing many operations in parallel, rather then serially.

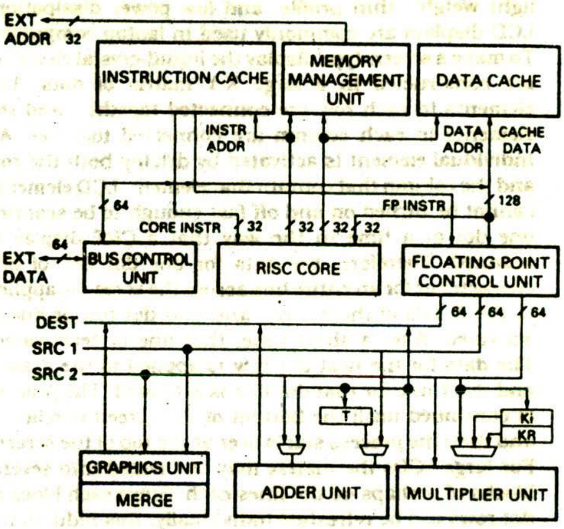


FIGURE 13-25 Block diagram of the Intel's i860™.

As you can see in Figure 13-25, the device contains a RISC processor core, an integer processor, a floating-point processor, and a graphics processor. The device also contains an instruction cache and a data cache. The separate caches mean that instructions and data can be read at the same time by a processor. Also, the 64-bit external data bus allows a 32-bit data word and a 32-bit instruction word to be read in at the same time.

The main processor in the i860™ is a *reduced instruction set computer* or *RISC*. As the name implies, this type processor has a very simple set of instructions which operate very fast. A RISC processor typically has only logical, simple arithmetic, shift, load, store, and jump instructions. When programming a RISC processor you use these simple instructions to "custom make" the more complex operations you need for a specific application. The advantage of a RISC approach over the *complex instruction set computer* or *CISC* approach of a processor such as the 8086 is that you don't have the overhead of, for example, string instructions which you may never use. RISC instructions typically execute in one or two clock cycles. Another advanced feature integrated in the i860™ is a complete memory-management unit shown at the top of Figure 13-25. In Chapter 15 we describe the operation of memory-management units.

Other Display Technologies

ALPHANUMERIC/GRAPHICS LCD DISPLAYS

In Chapter 9 we discussed the operation, advantages, and interfacing of LCD for displaying individual numbers and letters as individual digits. Because of their light weight, thin profile, and low power dissipation. LCD displays are commonly used in laptop computers. To make a screen-type display the liquid-crystal elements are constructed in a large X-Y matrix of dots. The elements in each row are connected together, and the elements in each column are connected tog ther. An individual element is activated by driving both the row and the column that contain that element. LCD elements cannot be turned on and off fast enough to be scanned one dot at a time in the way that a CRT display is scanned. Therefore, the data for one dot line of one character or for an entire line across the screen is applied to the X axis of the matrix, and that dot row or line is activated. After a short time, that line is deactivated. The data for the next dot row is applied to the X axis, and the Y line for that dot line is activated. The process is continued until the bottom of the screen is reached and then the process starts over at the top of the screen. For large LCDs the matrix may be divided into several blocks of perhaps 40 dot lines each. Since each block of dot rows can be refreshed individually, this reduces the speed at which each liquid-crystal element must be switched in order to keep the entire display refreshed. Large LCDs usually come with the multiplexing circuitry built in so that all you have to do is send the display data to the unit in the format specified by the manufacturer for that unit.

Most laptop computers in the past have used reflective type LCD displays because these displays use very low power. However, reflective displays have the disadvantages that they have low contrast, a relatively narrow viewing angle, and only monochrome displays. A common method of improving these displays has been to display different colors as different shades of gray. The shade of gray is determined by the duty cycle for which the pixel is activated. A device such as the 82C455 graphics controller from Chips and Technologies contains all the circuitry needed to interface with a VGA, EGA, or CGA LCD display with gray scaling.

To produce color displays some portable computers now use transmission-type LCD displays. In this type display the light from a strong fluorescent backlight is passed through the LCD elements and some filters to produce the desired display. As with color CRTs, a triad consisting of a red element, a green element, and a blue element makes up each pixel. The color of the pixel is determined by the relative intensity of the three elements.

In addition to their use as direct display devices transmission type LCD displays can be used to display the output of a computer on an overhead projector. A transparent frame containing the LCD panel and the interface electronics is simply placed on the overhead projector in place of the usual plastic overhead transparency. An example of this type unit is the PC Viewer from In Focus Systems; it produces a 640 × 480 × 16 color display.

The major disadvantages of the transmission-type LCDs currently are their high cost and the relatively large amount of power used by the backlight. The backlight power limits their use in battery-powered laptops.

PLASMA DISPLAYS

Another type display commonly used in portable computers is the plasma type. Plasma displays take advantage of the fact that some gases give off light when an electric current is passed through them. You have no doubt seen neon signs, which use this same principle.

A CRT plasma display consists of an X-Y matrix of pixels which contain neon gas between two electrodes in a tiny glass envelope. When a voltage is applied to both the X and the Y electrodes for a pixel, the pixel will light. A line on the display can be refreshed by applying the data to the X inputs for that line and applying a voltage to the Y input for that line.

Most plasma displays are orange because of the neon gas used, but different shades are commonly used to represent different colors. Color plasma displays are under development. Plasma displays typically have better contrast than reflective LCD, but they also require more power.

COMPUTER MICE AND TRACKBALLS

Figure 13-26*a* shows a common three-button mouse and Figure 13-26*b* shows a common trackball. As we're sure you know, devices such as these are commonly used to move the cursor around on a CRT screen to make drawings or execute commands by selecting a command from a menu. On the bottom of most mice there is a ball which rotates as you move the mouse around on your desk. As the ball rotates, it turns two optical encoder disks. One disk detects mouse motion in the X, or horizontal, direction and the other encoder detects motion in the Y, or vertical, direction. A trackball is essentially a mouse turned upside down so that you rotate the ball directly, instead of moving the mouse around to rotate the ball. The output data from a mouse or trackball consists of the condition of the switches and the amount of motion in the X and Y direction.

There are three major ways that mice and trackballs are interfaced to a computer. Serial mice connect to an RS-232-type serial port such as COM1 on the computer. Bus mice use an interface board which plugs into a slot in the motherboard of the computer. PS/2-type computers have a direct rear panel input especially designed for mice and other "pointer"-type devices.

When you buy a mouse or trackball, it usually comes with a diskful of programs and a fairly large instruction manual. Included on the disk for a mouse are two "drivers," typically called MOUSE.COM and MOUSE.SYS. You install one of these drivers to allow application programs to interface with the mouse. To install the standard mouse driver, MOUSE.COM, you simply copy the file to the DOS subdirectory on your hard disk and insert the command MOUSE in your AUTOEXEC.BAT file. To use the alternate driver, MOUSE.SYS, you copy this file to the DOS subdirectory on your hard disk and insert the statement device = mouse.sys in your CONFIG.SYS file. Either of these methods will load the mouse driver automatically when you boot up the system.

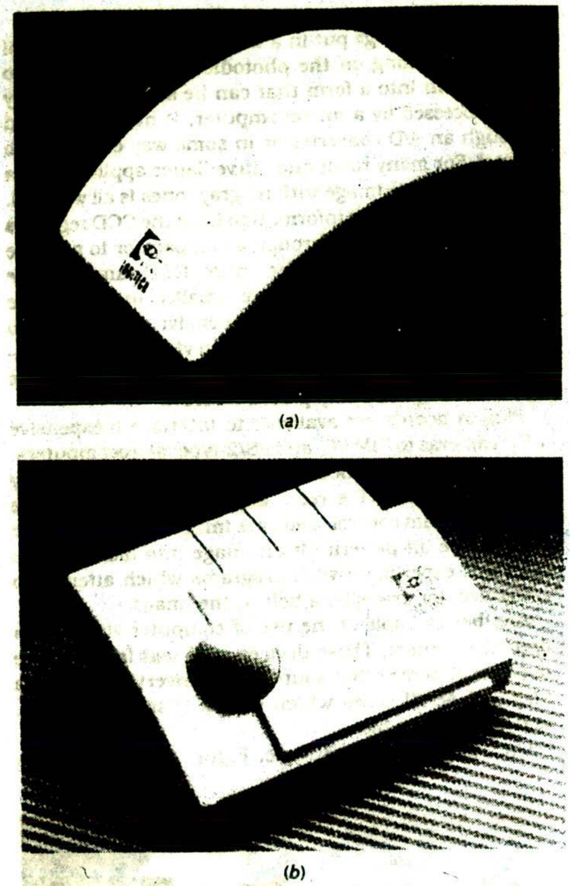


FIGURE 13-26 (a) Three-button mouse. (b) Trackball. (LOGITECH.)

In addition to the mouse drivers, the disk that comes with the mouse also contains a program which allows you to create pop-up menus containing a list of commands. To execute one of the commands in a pop-up menu, you move the mouse until a highlighted box appears on the desired command and then press a specified mouse key. For many people this sequence of operations is easier than typing in a command at the DOS prompt. To develop a menu you can choose from one of several menu templates supplied on the disk, or you can generate a custom menu format.

The question that may occur to you at this point is, How do I read mouse data to use in a program? If you are programming at the assembly language level, then the answer is to use the INT 33H procedure which is loaded into RAM as part of the mouse driver. As an introduction, Figure 13-27 shows some of the INT 33H subprocedures, the registers you use to pass parameters to these procedures, and the parameters that are passed back by the procedure. For more information on these consult the IBM Technical Reference Manual for the PS/2 Model 80 Microcomputer.

```
INTRODUCTION TO INT 33H MOUSE INTERFACE SUBPROCEDURES
AX     FUNCTION
1      Show visible pointer
2      Hide visible pointer

3     Get position and button status
      Returns: BX bit 0 = 1 - left
               BX bit 1 = 1 - right
               BX bit 2 = 1 - center
               CX = X coordinate
               DX = Y coordinate

4     Set pointer position
      CX = new horizontal position
      DX = new vertical position

11    Read mouse motion counters
      Returns: CX = horizontal mickeys count
               DX = vertical mickeys count

15    Set mickey/pixel ratio (sensitivity)
      CX = horizontal #mickeys/8 pixels
      DX = vertical #mickeys/8 pixels
```

FIGURE 13-27 Some INT 33H mouse-interface subprocedures.

For high-level-language interfacing with a mouse, some mouse manufacturers supply a library of mouse functions which you can call from your programs. The disk which comes with the Mouse systems mouse, for example, includes the file MSMOUSE.LIB, which contains a C function called mousec(). To use the mousec() function, you first declare four integer variables named m1, m2, m3, and m4 and then load these variables with the values that you want to pass to the AX, BX, CX, and DX registers, respectively, in the INT 33H procedure. You then call the function with the statement mousec (&m1, &m2, &m3, &m4);. The mousec function will put the mouse data in these variables in the same order that it would be returned in 8086 AX, BX, CX, and DX registers for a direct INT 33H call.

If you are programming in C and you do not have the library file containing the mousec function, you can use a function such as the Turbo C int86() to directly call the INT 33H procedure. The keyboard interface program in Figure 13-3 showed you an example of how to use the int86() function.

COMPUTER VISION

For many applications a microcomputer needs to be able to "see" its environment or perhaps a part that the machine it controls is working on. As part of a microcomputer-controlled security system, for example, we might want the microcomputer to "look" down a corridor to see if any intruders are present. In an automated factory application, we might want a microcomputer-controlled robot to "look" in a bin of parts, recognize a specified part, pick up the part, and mount the part on an engine being assembled. There are several mechanisms that can be used to allow a computer to see.

Cameras used in TV stations and for video recorders use a special vacuum tube called a *vidicon*. A light-sensitive coating on the inside of the face of the vidicon is swept horizontally and vertically by a beam of electrons. The beam is swept in the same way as the beam in a TV set displaying the picture will be swept. The amount of beam current that flows when the beam is at a particular spot on the vidicon is proportional to the intensity of the light that falls on that spot. The output signal from the vidicon for each scan line then is an analog signal proportional to the amount of light falling on the points along that scan line. In order to get this analog video information into a digital form that a computer can store and process, we have to pass it through an A/D converter. For a color camera we need an A/D converter on each of the three color signals. Each output value from an A/D converter then represents a dot of the picture. The number of bits of resolution in the A/D converter will determine the number of intensity levels stored for each dot.

Standard video cameras and the associated digitizing circuitry are relatively expensive, so they are not cost-effective for many applications. In cases where we don't need the resolution available from a standard video camera, we often use a CCD camera.

Charge-coupled devices or CCDs are constructed as long shift registers on semiconductor material. Figure 13-28 shows the structure for a CCD shift register section. As you can see, the structure consists of simply a *P*-type substrate, an insulating layer, and isolated gates. If a gate is made positive with respect to the substrate, a "potential well" is created under that gate. What this means is that if a charge of electrons is injected into the region under the gate, the charge will be held there. By applying a sequence of clock signals to the gates, this stored charge can be shifted along to the region under the next gate. In this way a CCD can function as an analog or a digital shift register.

To make an image sensor, several hundred CCD shift registers are built in parallel on the same chip. A photodiode is doped in under every other gate. When all the gates with photodiodes under them are made positive, potential wells are created. A camera lens is used to focus an image on the surface of the chip. Light shining on the photodiodes causes a charge proportional to the light intensity to be put in each well which has a diode. These charges can be shifted out to produce the dot-by-dot values for the scan lines of a picture. Improved performance can be gained by alternating nonlighted shift registers with the lighted ones. Information for a scan line is shifted in parallel from the lighted register to the dark and then shifted out serially.

The video information shifted out from a CCD register is in discrete samples, but these samples are analog because the charge put in a well is simply a function of the light shining on the photodiode. To get the video information into a form that can be stored in memory and processed by a microcomputer, it must be passed through an A/D converter or in some way converted to digital. For many robot and surveillance applications, a black-and-white image with no gray tones is all we need. In this case the video information from the CCD registers can simply be passed through a comparator to produce a 1 or a 0 for each dot of the image. CCD cameras have the advantages that they are smaller in size, more rugged, more sensitive, less expensive, and easier to interface to computer circuitry than vidicon-based cameras. For these reasons CCD cameras were used in the space telescope recently placed in orbit.

Plug-in boards are available to interface inexpensive CCD cameras to IBM PC- and PS/2-type microcomputers. With one of these boards installed, you can display images on the CRT screen, adjust display parameters under program control, and save images on a disk. Once you get the bit pattern for an image into memory, you can then experiment with programs which attempt to recognize, for example, a bolt in the image.

Another example of the use of computer vision is in *optical scanners*. These devices read text from a piece of paper or some other source and convert the text to a string of ASCII codes which can be displayed, edited, and written to a file.

On a more whimsical note, Figure 13-29 shows an

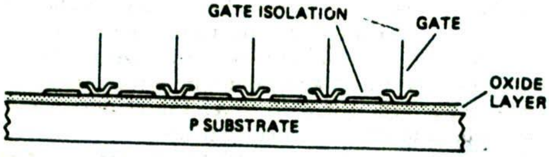


FIGURE 13-28 Structure for a CCD shift-register section.

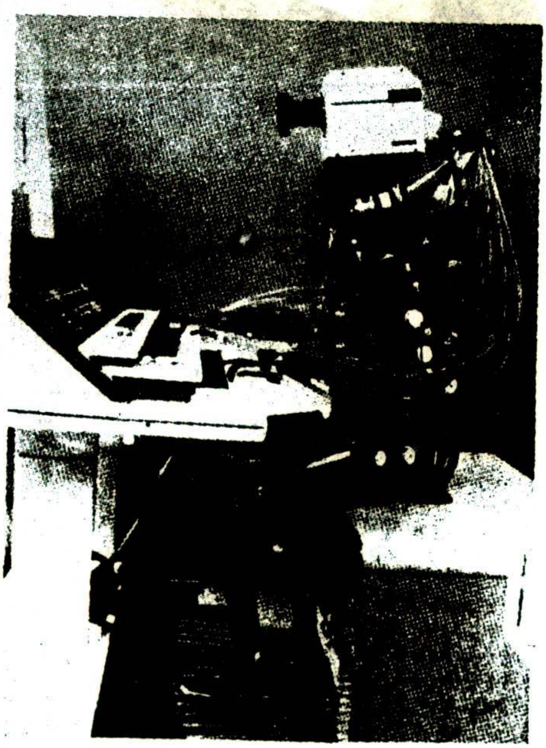


FIGURE 13-29 Sumitomo Electric Company robot playing an organ.

example of what a little vision can do for a robot. The Sumitomo Electric Company robot shown here can play an organ using both hands on the keys and both feet on the pedals. It can press up to 15 keys per second. The robot can play selections from memory when verbally told to do so. Using its vision it can read and play songs from standard sheet music. The robot uses seventeen 16-bit microprocessors and fifty 8-bit controllers to control all its activities.

If you think some about what is involved in recognizing complex visual shapes—in any of their possible orientations—with a computer program, it should give you a new appreciation for the pattern recognition capabilities of the human eye-brain system.

Another area where the human brain excels is in that of data storage. Only very recently have the devices used to store computer data approached the capacity of the human brain. In the next section we look at how some of these mass data storage systems operate and how they are interfaced to microcomputers.

MAGNETIC-DISK DATA-STORAGE SYSTEMS

The most common devices used for mass data storage are magnetic tape, floppy magnetic disks, hard magnetic disks, and optical disks. Magnetic tapes are used mostly for backup storage, because the access time to get to data stored in the middle of the tape is usually too long to be acceptable for general computing. Therefore, in this section we will concentrate mostly on the three types of disk storage.

Floppy-Disk Overview

Common sizes for floppy disks are 8, 5¼, and 3½ in. Figure 13-30*a* shows the flexible protective envelope used for 8 and 5¼-in. disks and Figure 13-30*b* shows the rigid plastic package used for the 3½-in. disks.

The disk itself is made of Mylar and coated with a magnetic material such as iron oxide or barium ferrite. The Mylar disk is only a few thousandths of an inch thick, thus the name floppy. When the disk is inserted in a drive unit, a spindle clamps in the large center hole or in the center hub and spins the disk at a constant speed of perhaps 300 or 360 rpm.

Data is stored on the disk in concentric, circular tracks. There is no standard number of tracks for any size disk. Older 8-in. disks have about 77 tracks/side, common 5¼-in. disks about 40 tracks/side, and the new 3½-in. disks about 80 tracks/side. Early single-sided drives recorded data tracks on only one side of the disk. Current double-sided disk drives store data on both sides of the disk.

Data is written to or read from a track with a read/write head such as that shown diagrammatically in Figure 13-31, page 466. During read and write operations the head is pressed against the disk through a slot in the envelope.

To write data on a track a current is passed through the coil in the head. This creates a magnetic flux in the iron core of the head. A gap in the iron core allows the magnetic flux to spill out and magnetize a section of the magnetic material along the track. Once a region on the track is magnetized in a particular direction, it retains that magnetism. The polarity of the magnetized region

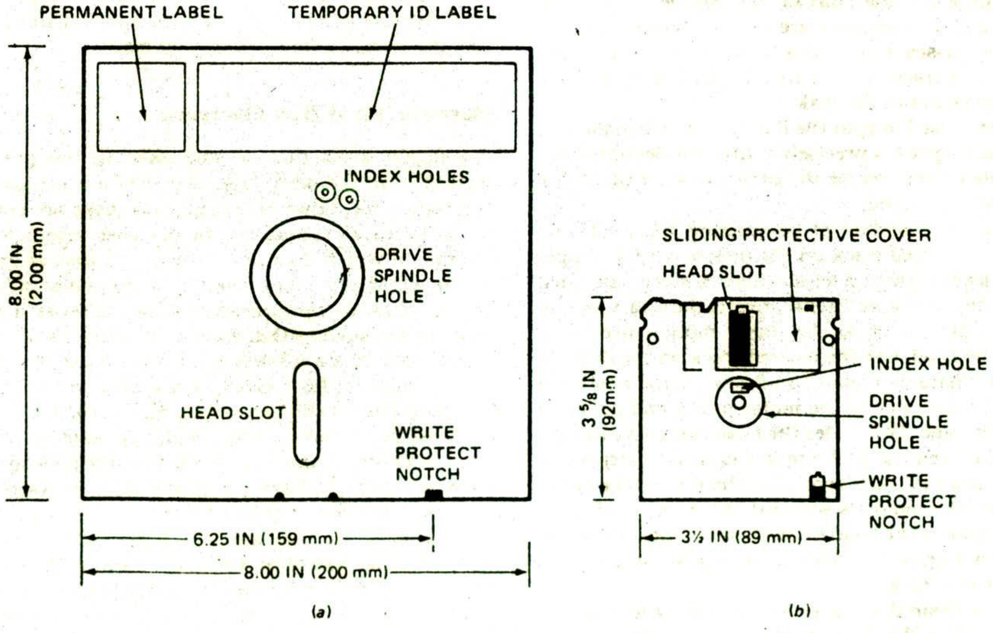


FIGURE 13-30 Common floppy-disk packages. (*a*) Package for 8- and 5¼-in. disks. (*b*) Hard plastic package used for 3½-in. disks.

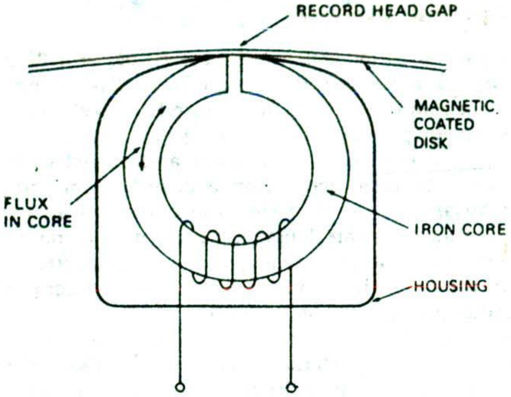


FIGURE 13-31 Diagram of read/write head used for magnetic-disk recording.

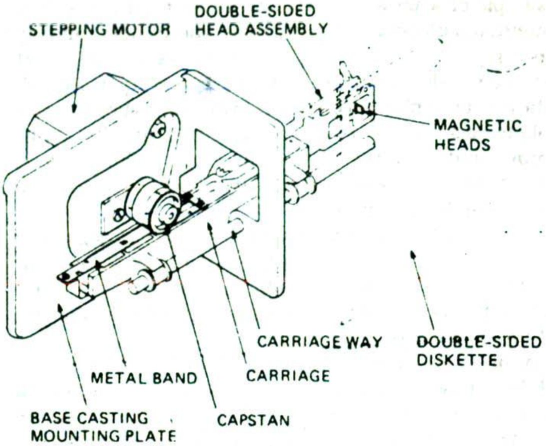


FIGURE 13-32 Common head-positioning mechanism for floppy-disk drive units.

is determined by the direction of the current through the coil. We will say more about this later.

Data can be read from the disk with the same head. Whenever the polarity of the magnetism along the track changes as the track passes over the gap in the read/write head, a small pulse of typically a few millivolts is induced in the coil. An amplifier and comparator convert this small signal to a standard logic level pulse.

The write-protect notch in a floppy disk envelope can be used to protect stored data from being written over, as do the knock-out plastic tabs on video tape cassettes. An LED and a phototransistor in the drive unit determine if the notch is present and enable the write circuits if it is.

On 8-in. and 5¼-in. disks, an index hole punched in the disk indicates the start of the recorded tracks. An LED and a phototransistor are used to detect when the index hole passes as the disk rotates. On 3½-in. disks the start of a track is indicated by the position of the hub in the center of the disk.

The motor used to spin the floppy disk is usually a dc motor whose speed is precisely controlled electronically. It takes about 250 ms for the motor to start up after a start-motor command.

One common method of positioning the read/write head over a desired track on the disk is with a stepper motor. A lead screw or a let-out-take-in steel band such as that shown in Figure 13-32 converts the rotary motion of the stepper motor to the linear motion needed to position the head over the desired track on the disk. As the stepper motor in Figure 13-32 rotates, the steel band is let out on one side of the motor pulley and pulled in on the other side. This slides the head along its carriage.

To find a given track, the motor is usually stepped to move the head to track zero near the outer edge of the disk. Then the motor is stepped the number of steps required to move the head to the desired track. It usually takes a few hundred milliseconds to position the head over a desired track.

Once the desired track is found, the head must be pressed against the disk or *loaded*. It takes about 50 ms to load the head and allow it time to settle against the disk.

If you add the time to start the motor, position the head over the desired track, and load the head, you can see that these operations take 100 to 500 ms, depending on the particular drive. When referring to disk drives, two different access times are usually given. One access time is the time required to get the head to the required track. This time is often called the *seek time*. The other access time is the time required to get to the first byte of a desired block of data on a track. This time is commonly called the *latency time*. For comparing the performance of drives, the average seek time is added to the average latency time to give an *average access time*. Average access times for currently available floppy disk drives range from 100 to 500 ms.

Magnetic Hard-Disk Overview

The floppy disks that we discussed in the previous section have the advantage that they are inexpensive and removable. However, because the disks are flexible, the data tracks cannot be put too close together, and the rate at which data can be read off a disk is limited by the fact that a floppy disk can be rotated at only 300 or 360 rpm. To solve these problems, we use a hard-disk system such as that shown in Figure 13-33.

The disks in a hard-disk system are made of a metal alloy, coated on both sides with a magnetic material. Common hard-disk sizes are 3½, 5¼, 8, 10½, 14, and 20 in. Most hard disks are permanently fastened in the drive mechanism and sealed in a dust-free package, but some systems do have removable disk packages. To increase the amount of storage per drive, several disks or "platters," as they are sometimes called, may be stacked with spacers between as shown in the Conner Peripherals' drive in Figure 13-33. A separate read-write head is used for each disk surface. On disk drives with more than one recording surface, the tracks are often called *cylinders* because if you mentally connect same numbered tracks on the two sides of a disk or on

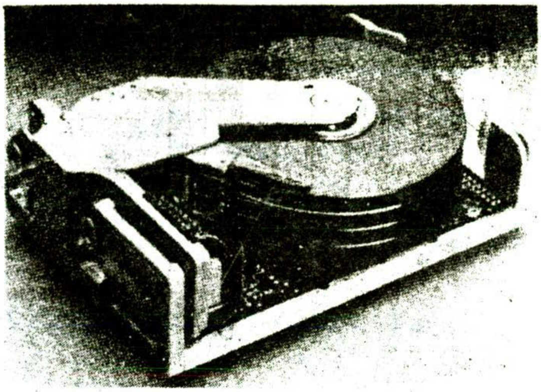


FIGURE 13-33 Cutaway photo of Conner Peripherals' CP3100 3-in., 100-Mbyte hard-disk drive.

different disks, the result is a cylinder. The cylinder number then is the same as the track number.

Hard disks are more dimensionally stable than floppies, so they can be spun faster. Large hard disks are rotated at about 1000 rpm and smaller hard disks are rotated at about 3600 rpm. Because the rotational speed is about 10 times that of a floppy disk, data is read out 10 times as fast, about 5 to 10 *Mbits/s*.

The dimensional stability of hard disks also means that tracks and the bits on the tracks can be put closer together. There are no standards for the number of tracks on a hard disk, but typically there are several hundred tracks on each side of a disk. The high rotational speed and the closely spaced tracks on a hard disk also produce much faster access times than those of floppy disks. The fastest currently available hard disks have average access times of less than 20 ms.

The high rotational speed of hard disks not only makes it possible to read and write data faster, it creates a thin cushion of air that floats the read-write head 10 to 100 μin. off the disk. Unless the head *crashes*, it never touches the recorded area of the disk, so disk and head wear are minimized. Hard disks must be kept in a dust-free environment because the diameter of dust and smoke particles may be 10 times the distance the head floats off the disk. If dust does get into a hard-disk system, the result will be the same as that which occurs when a plane does not fly high enough to get over a mountain. The head will crash and often destroy the data stored on the disk. When power to the drive is turned off, most hard disk drives retract the head to a *parking zone* where no data is recorded and lock the head in that position until power is restored.

In some early hard-disk drives the read-write heads were positioned over the desired track by a stepper motor and a band actuator, as shown in Figure 13-32. Most current hard-disk drives, however, use a *linear voice coil* mechanism or a *rotary voice coil* mechanism such as that shown in Figure 13-33 to position the read-write heads. This mechanism is essentially a linear motor. A feedback system adjusts the position of the head over the desired track until the strength of the signal read from the track is at its maximum.

Incidentally, hard-disk drives are sometimes called "Winchesters." Legend has it that the name came from an early IBM dual-drive unit with a planned storage of 30 Mbytes per drive. The 30-30 configuration apparently reminded someone of the famous rifle, and the name stuck.

Magnetic Disk Data Bit Formats

On a magnetic disk a "1" data bit is represented as a change in the polarity of the magnetism on the track. A "0" bit is represented as no change in the polarity of the magnetism. This form of recording is often called *nonreturn-to-zero* or NRZ recording because the magnetic field is never zero on a recorded track. Each point on the track is always magnetized in one direction or the other. The read head produces a signal when a region where the magnetic field changes passes over it.

Clock pulses are usually recorded along with the data bits on a track. The clock pulses read from the track are used to synchronize a phase-locked loop circuit. The output of the phase-locked loop is used to clock a D flip-flop at the center of the bit cell time where the data bits are written. The phase-locked loop is required to synchronize the read out circuits because the actual distance, and therefore time, between data bits read from an outer track is longer than it is for data bits read from an inner track. The phase-locked loop adjusts its frequency to that of the clock transitions and produces a signal which clocks the D flip-flop at the center of each bit time, regardless of the data rate. Recording clock information along with data information not only makes it possible to accurately read data from different tracks, but it also reduces the chances of a read error caused by small changes in disk speed.

Figure 13-34, page 468, shows the three common methods used to code data bits on magnetic disks. The top waveform in the figure shows how the example data bits are represented in a format called *frequency modulation*, *FM*, *F2F*, or *single-density* recording. Note the clock transition labeled C at the start of each bit cell in this format. These transitions represent the basic frequency. If the data bit in a cell time is a 1, the magnetic flux is changed again at the center of that bit time. If the data bit in a cell time is a 0, the magnetic flux is left the same at the center of that bit time. Putting in the 1 data transitions modifies the frequency, thus the name frequency modulation or F2F.

One major factor which determines how many data bits can be stored on a track is how close flux changes can be without interfering with each other. A disadvantage of standard F2F recording is that two transitions may be required to represent each data bit. A format which uses only half as many transitions to represent a given set of data bits is the *modified frequency modulation*, *MFM*, or *double-density* recording format shown as the second waveform in Figure 13-34. The basic principle of this format is that both clock transitions and 1 data transitions are used to keep the phase-locked loop and read circuitry synchronized. Clock transitions are not put in

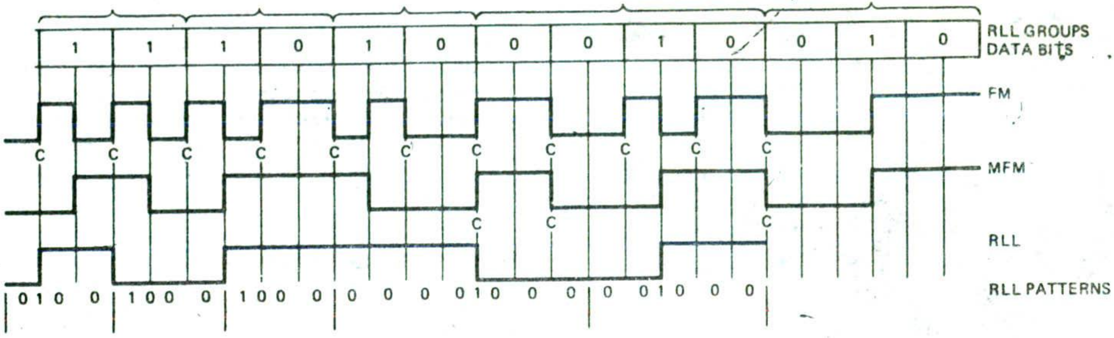


FIGURE 13-34 Comparison of FM, MFM, and RLL coding used for magnetic recording of digital data.

unless 1 transitions do not happen to come often enough in the data to keep the phase-locked loop synchronized. A clock transition will be put in at the start of the bit cell time only if the current data bit is a 0 and the previous data bit was a 0. If you work your way across the second waveform in Figure 13-34, you should see that where two 0's occur in the data sequence, a clock transition is written at the start of the bit cell for the second 0. The MFM waveform in Figure 13-34 is shown on the same scale as the F2F waveform, but since it contains only half as many transitions as the F2F waveform for the same data string, it can be written in half as much distance on a track. This means that twice as much data can be written on a track and explains why this coding is often called double density.

A still more efficient coding for recording data on floppy disks is the *RLL 2,7* format shown as the third waveform in Figure 13-34. In this format groups of data bits are represented by specific patterns of recorded transitions, as shown in Figure 13-35. The two 1's at the start of our data string, for example, are represented by the pattern 0100. To convert a data string to this coding, the data string is separated into groups chosen from the possibilities in the data column of Figure 13-35. The transition pattern which corresponds to that data bit combination is then written on the track. In MFM format there is at most one 0 between transitions, but in RLL 2,7 there are between two and seven 0's between transitions, depending on the sequence of data bits. As you can see in Figure 13-34, RLL requires considerably fewer transitions than MFM to represent the example data string. Fewer transitions mean that the data string can be written in a shorter section of the track or, in other words, more data can be stored on a given track. RLL coding typically increases the storage capacity about 40 percent over MFM coding.

DATA BIT GROUP	RLL 2, 7 CODE
1 0	1 0 0 0
1 1	0 1 0 0
0 0 0	1 0 0 1 0 0
0 1 0	0 0 1 0 0 0
0 1 1	0 0 0 1 0 0
0 0 1 0	0 0 0 0 1 0 0 0
0 0 1 1	0 0 1 0 0 1 0 0

FIGURE 13-35 RLL 2, 7 data bit groups.

Most of the magnetic floppy and hard disks of the last 15 years have used longitudinal recording. This means that the magnetic regions are oriented parallel to the disk surface along the track. Advances in read/write head design have made it possible to orient the magnetized regions vertically along the track. Vertical recording makes it possible to store several times as much data per track as can be stored with longitudinal recording. Toshiba and several other companies now market a disk drive which uses vertical recording to store 4 Mbytes of data on a single 3½-in. floppy disk such as that in Figure 13-30*b*. Some hard disks now available use vertical recording to store over a gigabyte of data in a single drive unit and transfer data at a rate of 3 Mbytes per second.

Magnetic-Disk Track Formats and Error Detection

In the preceding section we described the coding schemes commonly used to record data bits on floppy-disk or hard-disk tracks. The next level up from this is to show you the format in which blocks of data bytes are recorded along a track.

There are many slightly different formats commonly used to organize the data on a track, so we can't begin to show you all of them. However, to give you a general idea, Figure 13-36 shows an old standard, the IBM 3740 format, which is the basis of most current formats.

Each track on the disk is divided into sectors. In the 3740 format a track has three types of fields. An *index field* identifies the start of the track. *ID fields* contain the track and sector identification numbers for each of the 26 data sectors on the track. Each of the 26 sectors also contains a *data field* which consists of 128 bytes of data plus 2 bytes for a CRC error checking code. As you can see, besides the bytes used to store data, many bytes are used for track and sector identification, synchronization, error checking, and buffering between sectors. *Address marks* shown at several places in this

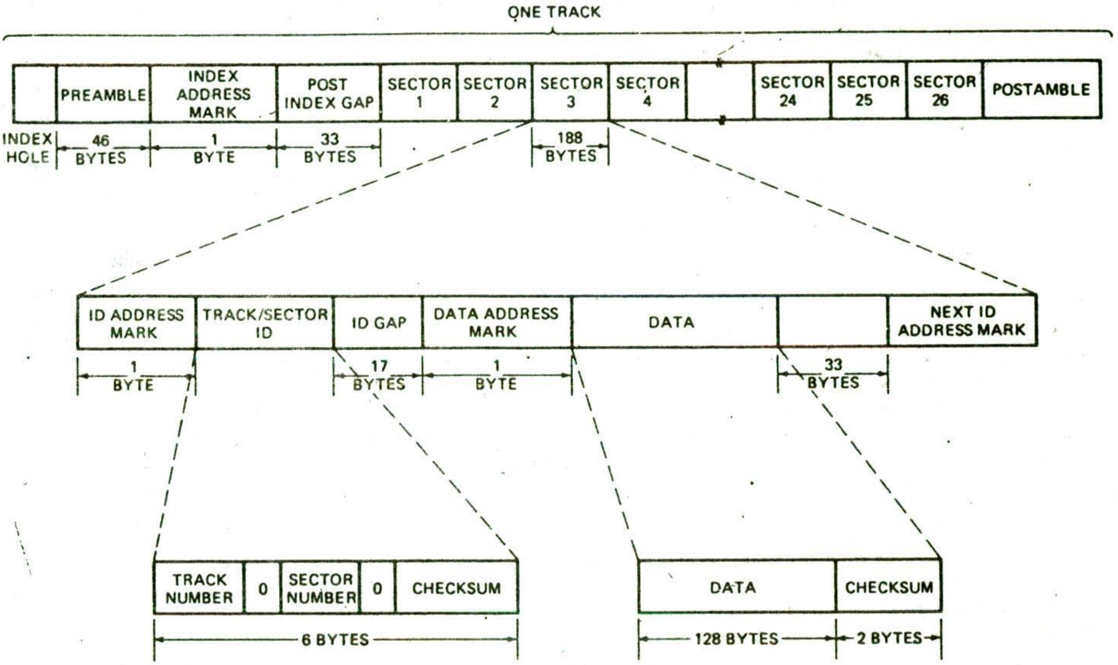


FIGURE 13-36 IBM 3740 floppy-disk soft-sectored track format.

format, for example, are used to identify the start of a field. Address marks, incidentally, have an extra clock pulse recorded with their D2 data bit so they can be distinguished from data bytes.

Two bytes at the end of each ID field and 2 bytes at the end of each data field are used to store *cyclic redundancy characters*. These are used to check for errors when the ID and the data are read out. One way the 2 CRC bytes can be produced is to treat the 128 data bytes as a single large binary number and divide this number by a constant. The 16-bit remainder from this division is written in after the data bytes as the CRC bytes. When the data bytes and the CRC bytes are read out, the CRC bytes are subtracted from the data string. The result is divided by the original constant. Since the original remainder has already been subtracted, the result of the division should be zero if the data was read out correctly. Higher-quality systems usually write data to a disk and immediately read it back to see if it was written correctly. If an error is detected, then another attempt to write can be made. If 10 write attempts are unsuccessful, then an error message can be sent to the CRT or the write can be directed to another sector on the disk.

The IBM 3740 format shown in Figure 13-36 is set up for single-density recording. An 8-in. disk in this format has one index track and 76 data tracks. Since each track has 26 sectors with 128 data bytes in each sector, the total is about 250 Kbytes. If double-density recording is used, the capacity increases to about 500 Kbytes. Using both sides of the disk increases the storage to about 1 Mbyte per disk. For reference, Figure 13-37 shows the number of tracks, number of sectors, and some other information for floppy disks commonly used with IBM PC and PS/2 computers. There are no real standards for the number of tracks and sectors on hard disks, but later we will give you a few examples.

Magnetic Disk Hardware Interfacing

AN 8272 FLOPPY DISK INTERFACE

As you can probably tell from the preceding discussion, writing data to a floppy disk and reading the data back requires coordination at several levels. One level is the drive motor and head-positioning signals. Another level is the actual writing and reading to the disk at the bit level. Still another level is interfacing with the rest of the circuitry of a microcomputer. Doing all this

	3 1/2-INCH DISKS		5 1/4-INCH DISKS	
	1.44MB HD	720K LD	1.2MB HD	360K DSDD
SECTORS PER TRACK	18	9	15	9
TOTAL NUMBER OF SECTORS	2,880	1,440	2,400	720
NUMBER OF TRACKS	80	80	80	40
SECTORS PER CLUSTER	1	2	1	2
ALLOWABLE ENTRIES IN ROOT DIRECTORY	224	112	224	112

FIGURE 13-37 Comparison of common floppy-disk tracks and sectors.

coordination is a full-time job, so we use a specially designed floppy-disk controller device to do it. As our example device here we will use the Intel 8272A controller, which is equivalent to the NEC μPD765A device used in many disk controller boards for IBM PC-type computers. We chose the 8272A because data sheets and application notes for it are available in Intel Microprocessor and Peripheral Handbook if you want more information than we have space for here.

Figure 11-5 showed you how an 8272A controller can be connected in an 8086-based microcomputer system to transfer data to and from a disk on a DMA basis. Now we want to take a closer look at the controller itself to show you the types of signals it produces and the operations it can perform.

To start, take a look at the block diagram of the 8272A in Figure 13-38. The signals along the left side of the diagram should be readily recognizable to you. The data bus lines, $\overline{RD}$, $\overline{WR}$, A0, RESET, and $\overline{CS}$ are the standard peripheral interface signals. The DRQ, $\overline{DACK}$, and INT signals are used for DMA transfer of data to and from the controller. To refresh your memory from Chapter 11, here's a review of how the DMA works for a read operation.

When a microcomputer program needs some data off the disk, it sends a series of command words to registers inside the controller. The controller then proceeds to find the specified track and sector on the disk. When the controller reads the first byte of data from a sector, it sends a DMA request, DRQ, signal to the DMA controller. The DMA controller sends a hold request signal to the HOLD input of the CPU. The CPU floats its buses and sends a hold-acknowledge signal to the DMA controller. The DMA controller then sends out the first transfer address on the bus and asserts the $\overline{DACK}$ input of the 8272 to tell it that the DMA transfer is underway. When the number of bytes specified in the DMA controller initialization has been transferred, the DMA controller asserts the TERMINAL COUNT input of the 8272. This causes the 8272 to assert its interrupt output signal, INT. The INT signal can be connected to a CPU or 8259A interrupt input to tell the CPU that the requested block of data has been read in from the disk to a buffer in memory. The process would proceed in a similar manner for a DMA write-to-disk operation.

Now let's work our way through the drive control signals shown in the lower right corner of the 8272 block diagram in Figure 13-38. Reading through our brief descriptions of these signals should give you a better idea of what is involved in the interfacing to the disk drive hardware. Note the direction of the arrow on each of these signals.

The READY input signal from the disk drive will be high if the drive is powered and ready to go. If, for

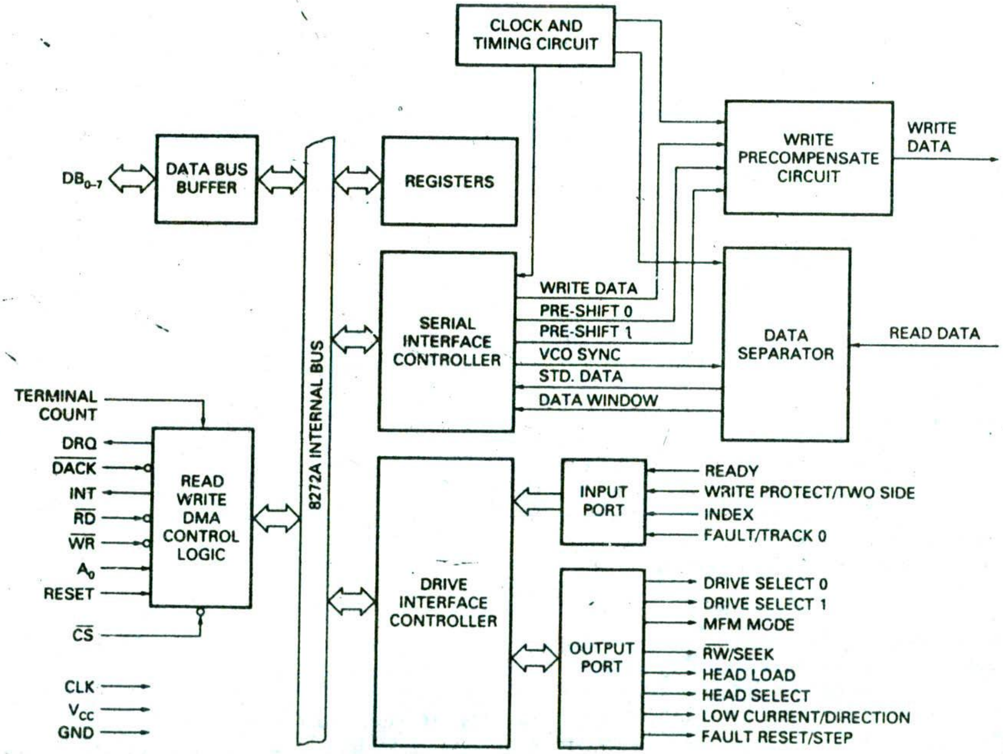


FIGURE 13-38 Block diagram of INTEL 8272A floppy-disk controller system. (*Intel Corporation.*)

example, you forget to close the disk-drive door, the READY signal will not be asserted.

The WRITE PROTECT/TWO SIDE signal indicates whether the write protect notch is covered when the drive is in the read or write mode. When the drive is operating in track-seek mode, this signal indicates whether the drive is two-sided or one-sided.

The INDEX signal will be pulsed when the index hole in the disk passes between the LED and phototransistor detector.

The FAULT/TRACK 0 signal indicates some disk-drive problem condition during a read/write operation. During a track-seek operation this signal will be asserted when the head is over track 0, the outermost track on the disk.

The DRIVE SELECT output signals, DS0 and DS1, from the controller are sent to an external decoder which uses these signals to produce an enable signal for one to four drives.

The MFM output signal will be asserted high if the controller is programmed for modified frequency modulation and low if the controller is programmed for standard frequency modulation (FM).

The $\overline{\text{RW}}$/SEEK signal is used to tell the drive to operate in read-write mode or in track-seek mode. Remember, some of the other controller signals have different meanings in the read-write mode than they do in the seek mode.

The HEAD LOAD signal is asserted by the controller to tell the drive hardware to put the read/write head in contact with the disk. When interfacing to a double-sided drive, the HEAD SELECT from the controller is used along with this signal to indicate which of the two heads should be loaded.

During write operations on inner tracks of the disk, the LOW CURRENT/DIRECTION signal is asserted by the controller. Because the bits are closer together on the inner tracks, the write current must be reduced to prevent recorded bits from splattering over each other. When executing a seek-track command this signal pin is used to tell the drive whether to step outward toward the edge of the disk or inward toward the center.

The FAULT RESET/STEP output signal is used to reset the fault flip-flop after a fault has been corrected when doing a read or write command. When the controller is carrying out a track-seek command, this pin is used to output the pulses which step the head from track to track.

Now that we have led you quickly through the drive interface signals, let's take a look at the 8272A signals used to read and write the actual clock and data bits on a track. To help with this, the upper right corner of Figure 13-38 shows a block diagram of the circuitry between these pins and the read/write head.

Remember from our discussion of FM, MFM, and RLL data formats that a phase-locked-loop circuit is required to tell the controller when to sample data bits in the input data stream. The V_{CO} SYNC signal from the controller tells an external phase-locked-loop circuit to synchronize its frequency with that of the clock and/data pulses being read off the disk. The output from the phase-locked-loop circuitry is a DATA WINDOW signal. This signal is sent to the controller to tell it where to find the data pulses in the data stream coming in on the READ DATA input.

For writing pulses to the disk, the story is a little more complex. External circuitry supplies a basic WR CLOCK signal. On its WR DATA pin the 8272 outputs the serial stream of clock bits and data bits that are to be written to the disk. During a write operation the 8272 asserts its WR ENABLE signal to turn on the external circuitry that actually sends this serial data to the read/write head. However, data bits written on a disk will tend to shift in position as they are read out. A 1 bit, for example, will tend to shift toward an adjacent 0 bit. This shift could cause errors in readout unless it were compensated for. The PRE-SHIFT 0 and PRE-SHIFT 1 signals from the controller go to external circuitry which shifts bits forward or backward as they are being written. The bits will then be in the correct position when read out.

8272 COMMANDS

The 8272 can execute 15 different commands. Each of these commands is sent to the data register in the controller as a series of bytes. After a command has been sent to the 8272, it carries out the command and returns the results to status registers in the 8272 and/or to the data register in the 8272. In programs you will almost always be interfacing with disks on a much higher level, but to give you an idea of the kinds of operations the 8272A controller can do, we list them here with a short description for each.

SENSE INTERRUPT STATUS—Return interrupt status information.

SPECIFY—Initialize head load time, head step time, DMA/non-DMA.

SENSE DRIVE STATUS—Return drive status information.

SENSE INTERRUPT STATUS—Poll the 8272 interrupt signal.

SEEK—Position read/write head over specified track.

RECALIBRATE—Position head over track 0.

FORMAT TRACK—Write ID field, gaps, and address marks on track.

READ DATA—Load head, read specified amount of data from sector.

READ DELETED DATA—Read data from sectors marked as deleted.

WRITE DATA—Load head, write data to specified sector.

WRITE DELETED DATA—Write deleted data address mark in sector.

READ TRACK—Load head, read all sectors on track.

READ ID—Return first ID field found on track.

SCAN EQUAL—Compare sector of data bytes read from disk with data bytes sent from CPU or DMA controller until strings match. Set bit in status register if match occurs.

SCAN HIGH OR EQUAL Set flag if data string from disk sector is greater than or equal to data string from CPU or DMA controller.

SCAN LOW OR EQUAL—Set flag if data string from disk sector is less than or equal to data string from CPU or DMA controller.

Working out a series of commands for a disk controller such as the 8272 on a bit-by-bit basis is quite tedious and time-consuming. Fortunately, you usually don't have to do this, because in most systems, you can use higher-level procedures to read from and write to a disk. In a later section we show you some of the software used to interface to disk drives.

ST-506, ESDI, AND SCSI HARD-DISK INTERFACES

The hardware interface for a hard disk is very similar to that for a floppy disk. Data is transferred to and from main memory on a DMA basis, as we described previously. However, in an attempt to maximize the rate of data transfer to and from the disk, several interface standards have developed. In order to understand these standards you first need to have an overview of how a hard disk is connected to microcomputer buses. The block diagram in Figure 13-39*a* shows how the hard disk in PC-type computers is usually connected. A board containing the hard-disk controller plugs into one of the expansion slots in the motherboard and a ribbon cable connects the controller to the hard disk.

One of the first standards for the interface connections between the controller card and the disk drive was the Seagate Technologies ST-506. This standard specified data and handshake signals very similar to those shown for the floppy interface in Figure 13-38. Standard 5.25-in. ST-506 hard disks use MFM recording with 17 sectors per track, 512 bytes per sector, and a rotation speed of 60 revolutions per second. The maximum rate at which data bits can be read from the track then is 60 tracks/s × 17 sectors/track × 512 bytes/sector × 8 bits/byte = 4,177,920 bits/s, or about 5 Mbits/s. The clocking of the ST-506 is set up to transfer data at a maximum rate of 5 Mbits/s, and this rate was more than adequate for early PC-type computers. In fact, it was necessary to use an *interleave factor* when writing data to the disk and reading data from the disk because the microprocessor and controller circuitry was not fast enough to read and transfer one sector directly after another. If the controller is programmed for an interleave factor of three, it will read a sector, skip over two sectors, and then read another sector. The skipped sectors give the controller time to transfer the data read from the first sector to main memory. Unfortunately, an interleave factor of three reduces the data transfer rate by a factor of three. As processors and controllers have become faster, it has become possible to decrease the interleave factor so that now an interleave factor of 1 is common. The limiting factor for data transfer then becomes the ST-506 transfer rate and the rate at which data bits can be read off the disk.

As we explained earlier, RLL encoding allows more data bits to be written on a track, so if RLL encoding is used, more sectors can be put on a track and the maximum data transfer rate increases to about 7.5 Mbits/s for an ST-506/RLL interface.

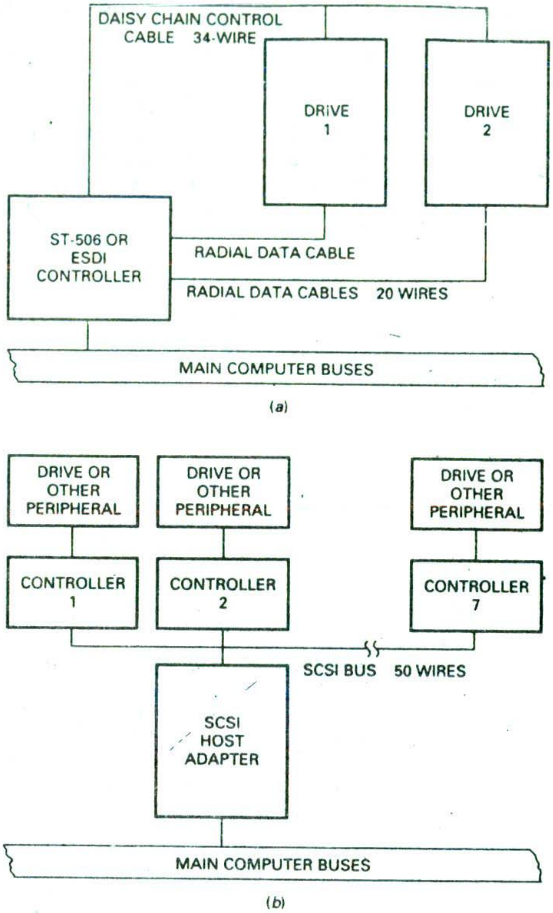

DAISY CHAIN CONTROL CABLE 34-WIRE
DRIVE 1
DRIVE 2
ST-506 OR ESDI CONTROLLER
RADIAL DATA CABLE
RADIAL DATA CABLES 20 WIRES
MAIN COMPUTER BUSES
(a)
DRIVE OR OTHER PERIPHERAL
DRIVE OR OTHER PERIPHERAL
DRIVE OR OTHER PERIPHERAL
CONTROLLER 1
CONTROLLER 2
CONTROLLER 7
SCSI BUS 50 WIRES
SCSI HOST ADAPTER
MAIN COMPUTER BUSES
(b)

FIGURE 13-39 Common hard-disk controller interface connections. (*a*) ST-506 or ESDI controller. (*b*) SCSI I/O bus.

The next evolutionary hard disk interface step was the *enhanced small device interface* (ESDI) standard. As shown in Figure 13-39*a*, an ESDI controller interfaces the system bus with hard-disk drives similarly to the way an ST-506 controller does. However, an ESDI controller can access up to seven hard drives using a daisy-chained control cable and individual data cables. Also, an ESDI controller sends higher-level commands to the drive than ST-506, so the drive must have more built-in "intelligence" to interpret these commands. A 10-MHz clock is used for the controller, so the maximum data transfer rate is 10 Mbits/s or 1.25 Mbytes/s. The ESDI standard allows communication with hard disks with maximums of 4096 cylinders, 16 heads, 256 sectors per track, and 4096 bytes per sector. The IBM PS/2 Model 80 uses an ESDI controller.

Another interface standard which was developed about the same time as ESDI is the small computer systems interface (SCSI), which is commonly pronounced "scuzzy." As shown in Figure 13-39*b*, this standard is very different from the ST-506 and ESDI, because it defines a separate I/O bus. Many different I/O devices such as hard disks, streaming tape drives, optical disk drives, and printers can be connected on this I/O bus. The SCSI host adapter converts operating system commands into SCSI bus commands. These commands are interpreted and carried out by the individual peripheral controllers. An ESDI controller, for example, might be used to interface the SCSI bus with a couple of hard drives. The question that may immediately come to mind here is, Why would anyone want to put the extra layer of hardware between the microcomputer bus and the hard drive controller? The answer to this question is that with a separate I/O bus, many data transfers can take place with very little effort on the part of the main microprocessor. For example, data can be transferred directly from a hard disk to a streaming tape backup on the SCSI bus without having to pass though the main microcomputer data bus. SCSI is designed to allow data transfer at up to 32 Mbits (4 Mbytes) per second. A newer standard, SCSI-II, is designed to allow data transfers at greater than 80 Mbits (10 Mbytes) per second. A still newer standard called enhanced intelligent peripheral interface (EIPI) is designed to allow data transfer at up to 50 Mbytes/s.

Disk Formatting

FLOPPY-DISK FORMATTING

As you probably well know by now, before you can store data on a new floppy disk you have to format it. To do this you use the DOS FORMAT command. The first operation this command performs is to establish a track and sector format such as that in Figure 13-36 on the disk. The second operation performed by the FORMAT command is to set up a boot record, file allocation table, and directory on the disk. Figure 13-40 shows how these are arranged, starting from track 0, sector 0.

The boot record in the first sector of the first track indicates whether the disk contains the DOS files needed to load DOS into RAM and run it. Loading DOS and running it are commonly referred to as "booting" the system.

The directory on the disk contains a 32-byte entry for each file. Let's take a quick look at the use of these bytes to get an overview of the directory information stored for each file.

Byte number (decimal)	
0–7	Filename
8–10	Filename extension
11	File attribute
	01H—read only
	02H—hidden file
	04H—system file
	08H—volume label in first 11 bytes, not filename
	10H—file is a subdirectory of files in lower level of hierarchical file tree
	20H—file has been written to and closed
12–21	Reserved
22–23	Time the file was created or last updated
24–25	Date the file was created or last updated
26–27	Starting cluster number
28–31	Size of the file in bytes

Most of these parameters should be familiar to you, but the term *cluster* may be new. DOS allocates disk space in clusters of one or more sectors. As shown in Figure 13-37, the number of sectors per cluster depends on the disk size and format. The file allocation table or FAT put on the disk during the FORMAT operation contains an entry for each cluster. The code stored in a FAT entry indicates whether the cluster is available, used, or defective. When you tell DOS to write a file to disk, it searches through the FAT until it finds a cluster which is marked as unused, writes the data to the cluster, and writes the code for used in the FAT entry. If the file is larger than one cluster, DOS searches the FAT until it finds another unused cluster, writes data to the cluster, and writes a used code in the FAT entry for the cluster. To establish a link with the first cluster of the file, DOS writes the number for the second cluster in the FAT entry for the first. The process continues until enough clusters are allocated to contain the file. To summarize, then, the file is stored as a chain of clusters and the FAT entry for each cluster contains the number of the next cluster in the chain. When DOS finishes writing the file to the disk, it updates the directory entry for the file with the time, date, and the number of the starting cluster for the file. Incidentally, DOS actually maintains two identical FATs to provide a backup in case one is damaged.

Also notice in the directory entry format shown here that an entry can represent a file or the name of a subdirectory. Each subdirectory can also refer directly to program or data files, or it can refer to a lower subdirectory. The point here is that this "tree" structure allows you to group similar files together and to avoid going through a long list of filenames to find a particular file you need. To get to a file in a lower-level directory, you simply specify the *path* to that file. The path is simply the series of directory names that you go through to get to that file.

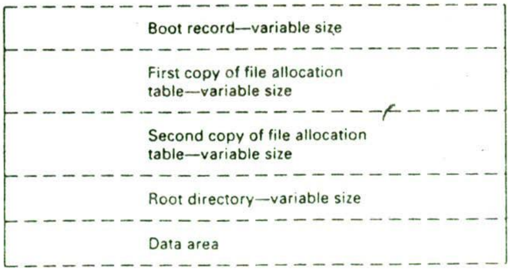

Boot record—variable size
First copy of file allocation table—variable size
Second copy of file allocation table—variable size
Root directory—variable size
Data area

FIGURE 13-40 DOS organization of boot record, FATs, directory, and data starting from track 0.

HARD-DISK FORMATTING

Formatting a hard disk usually now involves three different operations, low-level formatting, partitioning, and high-level formatting. To do the low-level format you use DEBUG to execute a program supplied by the manufacturer of the hard-disk controller card. The low-level format operation involves telling the controller the coding, the number of tracks, the number of sectors, the number of data bytes per sector, etc. In a low-level format you also specify the location(s) of any bad sector(s) on the disk. These bad sectors then will not be listed as available. Still another important value you specify in a low-level format is the interleave factor. As we explained before, the interleave factor is the number of sectors between consecutively numbered sectors on a track.

An important point about low-level formatting is that the format generated by a controller card from one manufacturer may not be the same as the format generated by one from another one. This means that if you move a hard disk from one controller card to another, you usually have to do a low-level format with the new controller before you can write to the disk.

To partition a hard disk, you use the DOS FDISK command. This command allows you to divide a large hard disk into as many as four logical drives. Each partition is assigned a drive identifier such as C:, D:, E:, etc. DOS versions before 4.0 limited the maximum size of each partition to 32 Mbytes, but later versions allow partitions of up to 2 Tbytes each. A sector called the partition table at the very start of the disk stores the start and stop cylinder numbers for each partition and a pointer to the partition that the system should boot from when the power is turned on.

To perform high-level formatting on a hard disk, you execute the DOS FORMAT command for each partition. The FORMAT command sets up the boot record, FAT, and directory for that partition in the same way as we described previously for a floppy. If the /s option is specified with the FORMAT command, the two system files needed to load the operating system from disk to RAM will automatically be copied to the disk. When the formatting is done, the rest of the DOS files are copied to the boot partition.

Disk-Drive Interface Software

BIOS-LEVEL INTERFACING

There are several different software levels at which you can interact with a disk drive. You can program directly at the controller level, but this is very tedious. Another approach is to use the BIOS INT 13H procedures to interface with a hard or floppy disk. The difficulty with this procedure is that you have to specify the particular track and sector(s) that you want to read or write and several other parameters. DOS function calls are much easier to use.

USING DOS FUNCTION CALLS

A large part of an operating system such as DOS is a collection of procedures which perform tasks such as formatting disks, creating disk files, writing data to files, reading data from files, and communicating with system peripherals such as modems and printers. DOS allows you to call these procedures from your programs with an INT 21H instruction, similar to the way you call BIOS procedures.

Each DOS function (procedure) has an identification number. To call a DOS function you put the function number in the AH register, put any required parameters in the specified registers, and then execute the INT 21H instruction. As a first example, DOS function call 40H can be used to print a string. To use this procedure, set up the registers as follows:

1. Load the function number, 40H, into the AH register.
2. Load the DS register with the segment base of the segment which contains the string.
3. Load the DX register with the offset of the start of the string.
4. Load the CX register with the number of bytes in the string.
5. Load the BX register with 0004H, the fixed "file handle" for the printer.

Then, to call the DOS procedure, execute the INT 21H instruction. Note that the DOS function allows you to send an entire string to the printer, rather than just a single character at a time as the BIOS INT 17H does.

As another example, the DOS 0AH function will read in a string from the keyboard and put the string in a buffer pointed to by DS:DX. Characters will also be displayed on the CRT as they are entered on the keyboard. The function terminates when a carriage return is entered. To use this function, first set up a buffer in the data segment with the DB directive. The first byte of the buffer must contain the maximum number of bytes the buffer can hold. The 0AH call will return the actual number of characters read in the second byte. The function does not require you to pass it a file handle, because the file handle is implied in the function.

You can also exit from one of your programs and return to the DOS command level using the DOS 4AH function. To do this, load AL with 00 and AH with 4CH and then execute the INT 21H instruction.

The main feature of DOS function calls that we want to discuss here, however, is working with disk files. Many disk operating systems and earlier versions of PC DOS require you to construct a *file control block* or FCB in order to access disk files from your programs. The format of a file control block differs from system to system, but basically the FCB must contain, among other things, the name of the file, the length of the file, the file attribute, and information about the blocks in the file. Version 2.0 and later versions of PC DOS simplify calling DOS file-handling procedures by letting you refer to a file with a single 16-bit number called a *file handle* or *token*. You simply put the file handle for a file you want to access in a specified register and call the DOS procedure which performs the desired action on that file. DOS then constructs the FCB needed to access the file. The question that may occur to you at this point is, How do I know what the file handle is for a file I want to access on a disk? The answer is that to get the file handle for a disk file, you simply call a DOS procedure which returns the file handle in a register. You can then pass the file handle to the procedure that you want to call to access the file. The point here is that file handles make it easy for you to access files.

DOS extends the concept of a file to include any device that can input or output data. DOS treats external devices such as printers, the keyboard, and the CRT as files for read and write operations. These devices are assigned fixed file handles by DOS as follows: 0000—keyboard, 0001—CRT, 0002—error output to CRT, 0003—serial port, 0004—printer. The significance of this is that you can use the same DOS function call to write the data in a buffer to a disk file, to the CRT screen, or to a printer. Also, a stream of data can be *redirected* from one file (device) to another. The DOS command DIR A: > LPT1, for example, will redirect the stream of data produced by the DIR command to the printer instead of sending it to the CRT, which is the normal output device for the DIR command.

As a final example here, Figure 13-41 shows you how DOS function calls can be used to open a file, read data from a file into a buffer in memory, and close the file. Opening a file means copying the file parameters from the directory to a file control block in memory and marking the file as open. Closing a file means updating the directory information for the file and marking the file closed. To open a file and get the file handle, we use DOS function call 3DH. For this call DS:DX must point to the start of an *ASCIIZ* string which contains the disk drive number, the path, and the filename. An ASCIIZ string is a string of ASCII characters which has a byte of all 0's as its last byte. Also, AL must contain an access code which indicates the type of operation that you want to perform on the file. Use an access code of 00 for read only, 01 for write only, and 02 for read and write. Again, to actually call the function, you load 3DH into AH and execute the INT 21H instruction. The handle for the opened file is returned in the BX register. The first part of Figure 13-41 shows how these pieces are put together.

```
;8086 PROGRAM FRAGMENT
;ABSTRACT : This code shows how to use DOS functions
           ; to open a file, read the file contents
           ; into a buffer in memory, and close the file
; Point at start of buffer containing file name
MOV  DX, OFFSET FILE_NAME
MOV  AL, 00                  ; open file for read
MOV  AH, 3DH                 ; and get file handle
INT  21H
MOV  BX, AX                  ; save file handle in BX
PUSH BX                      ; and push for future use
MOV  CX, 2048                ; set up maximum read
MOV  DX, OFFSET FILE_BUF     ; point at memory buffer
                             ; reserved for disk file
                             ; contents
MOV  AH, 3FH                 ; read disk file
INT  21H
POP  BX              ; get back file handle for close
PUSH AX              ; save file length returned by
                     ; 3FH function call
MOV  AH, 3EH         ; close disk file
INT  21H
; use the file now stored in memory
```

FIGURE 13-41 Using DOS function calls to open and read a file.

To read a file we use function call 3FH. For this call BX must contain the file handle and CX the number of bytes to read from the file. DS:DX must point to the buffer location in RAM that the data from the file will be read into. To do the actual call we load 3FH into AH and do an INT 21H instruction. After the file is read, AX contains the number of bytes actually read from the file.

To close the file we load function number 3EH into AH, load the file handle into BX, and execute the INT 21H instruction. The last half of Figure 13-41 shows the instructions you can use to read and close a file. Consult the IBM DOS Technical Reference Manual for the details of all of the available function calls.

DISK INTERFACING IN C

Interfacing with disk drives in C is in some ways very similar to using the DOS function calls as we described in the preceding section. In C as in DOS any device that can input or output data is referred to as a file. To make it easy for programmers to interface with this variety of files ANSI C buffers the files, similar to the way DOS does. Remember that DOS function calls buffer you from the track and sector details by allowing you to refer to a file with a simple file handle. The buffering also produces a uniform data stream regardless of the physical device characteristics. As we said above, this means that you can use the same DOS function call to send the data stream to any one of several different devices.

The predefined C functions for buffered I/O also allow you to work with a data stream in your programs instead of having to worry about the characteristics of the actual physical device or file. To refer to a file C uses a special pointer of type FILE. Type FILE is defined in stdio.h as a pointer to a data structure which is essentially the template for a file control block. To declare a file pointer for use in your program, you use a statement such as FILE *fp;. When you open a file you associate the pointer fp with the file, and for any further interactions with the file you can use fp, just as you use the file handle in DOS function calls. To give you a better idea of how this works, Figure 13-42, page 476, shows a program which opens a file for write, writes a line of text to the file, and then closes the file. The program then opens the file for read, writes the contents of the file to the screen, writes the contents of the file to the printer, and again closes the file. Note that this program requires very few statements to do a considerable amount of work.

To get you used to the way professional C programmers write code, we have included a few statements which contain several actions. The key to interpreting statements such as these, remember, is to start with the innermost parentheses and work your way out from there.

```
/* C PROGRAM F13-42.C */

#include<stdio.h>
#include<dos.h>
#include<conio.h>

main()
{
FILE *fp;
char filename[32];
char ch;
char textbuf[100];
char *tp =textbuf;
int count;

/* Open file, write line of text to it, close file */

printf("\n Please enter name of file you want to create.\n");
gets(filename);
if((fp=fopen(filename, "wt"))==0) /* fopen returns 0 if error */
        {
        perror(filename);    /* if error, print error message */
        exit();
        }
printf("Enter a line of text.\n");
gets(tp);                      /* read string into edit buffer */
count = strlen(tp);            /* determine number of bytes */
fwrite(tp,1,count,fp);         /* copy buffer to file */
fclose(fp);                    /* close file */

/* Open file for read and display file contents on screen */

if((fp=fopen(filename,"rt"))==0) /* open for read, check for error */
        {
        perror(filename);  /* if error, print error message */
        exit();
        }
while(!feof(fp))               /* while not end of file, read */
        fputc(fgetc(fp),stdout); /* file and send to screen */

/* Send contents of file to printer */

rewind(fp);     /* reset file pointer to start of file buffer */

while(!feof(fp))  /* send characters from buffer to printer */
        fprintf(stdprn, "%c", fgetc(fp));

fprintf(stdprn, "\n"); /* carriage return to printer */

fclose(fp);    /* close file */
exit();
}
```

FIGURE 13-42 C program which uses predefined functions to perform file operations.

The program in Figure 13-42 first declares a FILE pointer, as we described earlier, and then declares an array to contain a user-entered file name, and an array to contain a user entered line of text. The program then prompts the user to enter a file name and reads the entered filename into the array named filename. After this the action gets more interesting.

The fp = fopen(filename, "wt") part of the if statement opens the file named filename for write text operations and initializes the file pointer fp to point to the file control block for that file. In programming jargon we say that we have opened a stream named fp. The if(= = 0) part of this statement compares the value assigned to fp with 00. A value of 0 for fp indicates that some error occurred when the attempt was made to open the file. The drive door, for example, might have been open.

If an error occurred, we call the perror function, which determines the error that occurred and prints an appropriate error message to the screen. The exit function then returns execution to DOS.

If the file was opened without errors, we then prompt the user to enter a line of text and read the text into the array pointed to by tp. We then use the strlen function to determine how many bytes are in the entered string, so that we can pass this value to the fwrite function. The arguments you pass to the fwrite() function are a pointer to the array you want to write to the disk, the number of bytes in each data item in the array, the number of elements in the array, and the file pointer which identifies the file. The fwrite function actually writes the contents of the array to a buffer maintained by the buffering software, and this software takes care of actually writing the data to the disk. The fclose(fp) function closes the stream to this buffer, writes any data remaining in the buffer to the file, and closes the file.

In the next section of the program we open the file for read by passing the "rt" string to the fopen function. Again we print an error message if the file-open operation was unsuccessful. We then use a while loop to read characters from the fp stream and send them to the stdout stream until the end of file character is detected. The fgetc(fp) function reads a character from the fp stream. The character returned by fgetc() is passed to the fputc() function, along with the destination stream of stdout. Stdout is one of the predefined streams that is opened automatically by the C startup code. The predefined streams are stdin, stdout, errout, and stdaux, which refer to the CRT, and stdprn, which refers to the default printer. Now let's look at how we can send the contents of the file to the printer.

To keep track of the current location in a stream, the C buffering system maintains a stream pointer. After reading the characters in from the file and sending the characters to the CRT, the stream pointer for fp will be pointing at the end of the file. Before we can perform any other operation on this stream, we have to reset the stream pointer to the start of the file. The rewind(fp) function call does this.

The final part of the program in Figure 13-42 uses another while loop to read characters from the fp stream and send them to the printer until the end of file character is found. The arguments you pass to the fprintf() function are the destination stream, a format specifier, and the value of the variable you want to print. In this call the value to be printed is the value returned from the fp stream by the fgetc(fp) function call. Finally, we close the stream and the file with the fclose() function. An exercise in the accompanying lab manual gives you a further chance to work with these operations.

RAM DISKS

The VDISK command found in DOS versions 3.3 and later allows you to set aside an area of RAM in such a way that it appears to DOS as simply another disk drive. In a computer that has actual drives A:, B:, and C:, you can create a RAM or virtual drive which you access as D:. You can copy files to and from this RAM disk by name just as you would for any other drive. Here's an example of why you might want to set up a virtual drive.

When you load a large program such as Wordstar into memory to run it, the basic program and some commands are loaded, but some of the program remains on disk. This is done so that the program will run in systems that do not have a large amount of memory. When you execute a command that has not been loaded into memory, the code for that command is read from the disk and executed. If you have enough memory in your system, you can create a RAM drive and copy all the Wordstar files to that drive. The commands can then be accessed much faster because there is no mechanical access time as there is with an actual disk. The advantage of configuring the RAM as a disk drive is that the software can access it just as if it were on an actual disk.

DISK CACHES

Another commonly seen term in current computer periodicals is *disk cache*. A disk cache functions similarly to the RAM cache we discussed in Chapter 11. The principle of a RAM cache, remember, is to keep often-used sections of code and data in a fast SRAM cache where it can be accessed without wait states.

The DOS *BUFFERS command*, which you usually put in your CONFIG.SYS file with a statement such as BUFFERS=20, sets aside RAM to hold data read in from disk files. Each buffer created with this command contains 528 bytes. The problem with this approach is that if you create too many buffers, the overhead of determining if a desired file section is present becomes too long and performance actually decreases. Another problem with the BUFFERS approach is that only the requested data is read into the buffers.

Programs such as IBMCACHE which come with the PS/2 50, 60, and 80 computers allow you to set up a separate block of 16 Kbytes to 512 Kbytes as a cache for data read from disk. When disk read occurs, all or at least a large part of the file can be read into the disk cache. This reduces the number of disk accesses required and makes "disk-intensive" programs execute two or three times faster. When a program writes data to a file that is in the cache, the data is written to the cache and to the actual file on the disk, so that the data will not be lost in case of a power failure.

Hard-Disk Backup Storage

To prevent data loss in the event of a head crash, hard disk files are backed up on some other medium such as floppy disks or magnetic tape. The difficulty with using floppy disks for backup is the number of disks required. Backing up a 70-Mbyte hard disk with 1.2-Mbyte floppies requires about 60 disks and considerable time shoving disks in and out. Most systems with large hard disks now use a high-speed magnetic tape system for backup. A typical *streaming tape* system, as these high-speed systems are often called, can dump or load the entire contents of a large hard disk to a single tape in a few minutes.

OPTICAL DISK DATA STORAGE

The same optical disk technology used to store audio on compact disks can be used to store very large quantities of digital data for computers. One unit now available, the Maxtor Tahiti I, for example, stores up to a total of 1 Gbyte (1000 Mbytes) of data on a single, removable 5¼-in. disk. This amount of storage corresponds to about 400,000 pages of text. Besides their ability to store large amounts of data, optical disks have the advantages that they are relatively inexpensive and immune to dust, and most are removable. Also, since data is written on the disk and read off the disk with the light from a tiny laser diode, the read/write head does not have to touch the disk. The laser head is held in position about 0.1 in. away from the disk, so there is no disk wear. Also, the increased head spacing means that the head will not crash on small dust particles and destroy the recorded data as it can with magnetic hard disks.

The disk sizes now available in different systems are 3.5, 4.72 (the compact audio disc size), 5.25, 12, and 14 in. Data storage per disk ranges from 60 Mbytes to several gigabytes. The actual drive and head-positioning mechanisms for optical disk drives are very similar to those for magnetic hard-disk drives. A feedback system is used to precisely control the speed of the motor which rotates the disk. Some units spin the disk at a constant angular velocity (CAV) in the range of 700 to 3000 rpm. Other systems such as those based on the compact audio (CD) format adjust the rotational speed of the disk so that the track passes under the head with a constant linear velocity (CLV). With CLV the disk is rotated slower when reading outer tracks.

Some optical disk systems record data in concentric circular tracks as magnetic disks do. Other systems, such as the CD disk systems, record data on a single spiral track in the same way a phonograph record does. A linear voice coil mechanism with feedback control is used to precisely position the read head over a desired track. The head positioning must be very precise, because the tracks on an optical disk are very narrow and very close together. The tracks are typically about 20 μin. wide and about 70 μin. between centers. This spacing allows tens of thousands of tracks to be put on a disk.

Optical disk systems are available in three basic types: read only, write once/read many, and read/write.

Read-only systems allow only prerecorded disks to be read out. A disk which can only be read from is often called an *optical ROM* or *OROM*. Examples of this type are the 4.7-in. compact audio disks and the optical disk encyclopedias.

Write once/read many or *WORM* systems allow you to write data to a disk, but once the data is written, it cannot be erased or changed. The stored data can be read out as many times as desired.

Erasable optical or EO systems allow you to erase recorded data and write new data on a disk. The recording materials and the recording methods are different for these different types of systems.

Disks used for read only and write once/read many systems are coated with a substance which is altered when a high-intensity laser beam is focused on it with a lens. The principle here is similar to using a magnifying glass to burn holes in paper, as you may have done in your earlier days. In some systems the focused laser light produces tiny pits along a track to represent 1's. In other systems a special metal coating is applied to the disk over a plastic polymer layer. When the laser beam is focused on a spot on the metal, heat is transferred to the polymer, causing it to give off a gas. The gas given off produces a microscopic bubble at that spot on the thin metal coating to represent a stored 1. Both of these recording mechanisms are irreversible, so once written, the data can only be read. Data can be read from this type of disk using the same laser diode used for recording, but at reduced power. A system might for example use 25 mW for writing, but only 5 mW for reading.

To read the data from the disk, the laser beam is focused on the track and a photodiode is used to detect the beam reflected from the data track. A pit or bubble on the track will spread out the laser beam light so that very little of it reaches the photodiode. A spot on the track with no pit or bubble will reflect light to the photodiode. Read-only and write-once systems are less expensive than read/write systems, and for many data-storage applications the inability to erase and rerecord is not a major disadvantage. One example of a WORM optical drive is the Control Data Corporation LaserDrive 510, which stores 654 Mbytes on a removable ANSI/ISO standard 5¼-in. disk cartridge.

Most current read/write optical disk systems use disks coated with an exotic metal alloy which has the required magnetic properties. The read/write head in this type of system has a laser diode and a coil of wire. A current is passed through the coil to produce a magnetic field perpendicular to the disk. At room temperature the applied vertical magnetic field is not strong enough to change the horizontal magnetization present on the disk. To record a 1 at a spot in a data track, a pulse of light from the laser diode is used to heat up that spot. Heating the spot makes it possible for the applied magnetic field to flip the magnetic domains around at that spot and create a tiny vertical magnet. This is called *magneto-optical* or *MO recording*.

To read data from this magneto-optical type disk, polarized laser light is focused on the track. When the polarized light reflects from one of the tiny vertical magnets representing a 1, its plane of polarization is rotated a few degrees. Special optical circuitry can detect this shift and convert the reflections from a data track to a data stream of 1's and 0's. A bit is erased by turning off the vertical magnetic field and heating the spot corresponding to that bit with the laser. When heated with no field present, the magnetism of the spot will flip around in line with the horizontal field on the disk. One example of a currently available read/write optical drive that uses MO recording is the Maxtor Corporation Tahiti I, which stores about 600 Mbytes on an ANSI/ISO standard 5¼-in. disk cartridge or 1 Gbyte on a special 5¼-in. cartridge. The Tahiti I has an average access time of 30 ms, which compares favorably with the 16- to 25-ms access times of the fastest current hard-disk drives.

The maximum data transfer rate for the Tahiti I is 10 Mbits/s, which is the same order of magnitude as the transfer rate for the leading hard disk units. Incidentally, most optical disk drives use the SCSI interface we described previously.

The amount of data storage on one optical disk is impressive, but to store even more data there are now available several "jukebox" optical disk systems, which can hold up to 256 removable disks. Typically, it takes only a few seconds to load a desired disk into the actual drive so it can be accessed. Optical disks have the further advantage that the disk cartridge is easily removable and can be locked away for safety and security purposes.

The potentially low cost of a few cents per megabyte and the hundreds of gigabytes of data storage possible for optical disk systems may change the whole way our society transfers and processes information. The contents of a sizable library, for example, can be stored on a few disks. Likewise, the entire financial records of a large company can be kept on a single disk. "Expert" systems for medical diagnosis or legal defense can use a massive data base stored on disk to do a more thorough analysis. Engineering workstations can use optical disks to store data sheets, drawings, graphics, or IC-mask layouts. The point here is that optical disks bring directly to your desktop computer a massive data base that previously was available only through a link to large mainframe computers or, in many cases, was not available at all. The large data storage capacity of optical disks also make them useful for a system called Digital Video Interactive, which we discuss in the last section of this chapter.

PRINTER MECHANISMS AND INTERFACING

Many different mechanisms and techniques are used to produce printouts or "hard" copies of programs and data. This section is intended to give you an overview of the operation and trade-offs of some of the common printer mechanisms. We start with those that mechanically hit the paper in some way.

Dot-Matrix Impact Print Mechanisms

Figure 13-43 shows an impact-type dot-matrix print head. Thin print wires driven by solenoids at the rear of the print head hit the ribbon against the paper to produce dots. The print wires are arranged in a vertical column so that characters are printed out one dot column at a time as the print head is moved across a line of characters. As we mentioned in an earlier chapter, a stepper motor is commonly used to move the print head across the paper, and another stepper motor is used to advance the paper to the next character row.

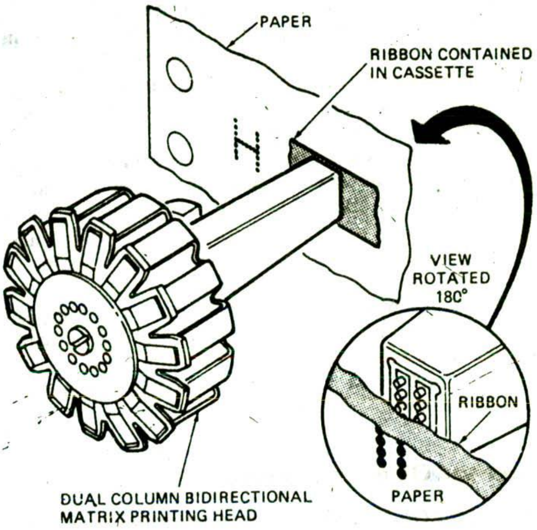

PAPER
RIBBON CONTAINED IN CASSETTE
VIEW ROTATED 180°
RIBBON
PAPER
DUAL COLUMN BIDIRECTIONAL MATRIX PRINTING HEAD

FIGURE 13-43 Impact dot-matrix printer mechanism. (*Courtesy DATAPRODUCTS Corporation.*)

Early dot-matrix print heads had only seven print wires, so print quality of these units was not too good. Currently available dot-matrix printers use 9, 14, 18, or even 24 print wires in the print head. Using a large number of print wires and/or printing a line twice, with the dots for the second printing offset slightly from those of the first, produces "letter-quality" print. Dot-matrix printers can also print graphics. To do this the dot pattern for each column of dots is sent out to the print-head solenoids as the print head is moved across the paper. The principle is similar to the way we produce bit-mapped raster graphics on a CRT screen. By using different-color ribbons and making several passes across a line, some dot-matrix impact printers allow you to print color graphics. Most dot-matrix printers now contain one or more microprocessors to control all this.

Print speeds for dot-matrix impact printers range up to 350 cps. Some units allow you to use a low-resolution mode of 200 cps for rough drafts, a medium resolution mode of 100 cps for finish copy, or 50 cps for near-letter-quality printing. The advantages of dot-matrix impact printers are their relatively low cost and their ability to change fonts or print graphics under program control.

Dot-Matrix Thermal Print Mechanisms

Most thermal printers require paper which has a special heat-sensitive coating. When a spot on this special paper is heated, the spot turns dark. Characters or graphics are printed with a matrix of dots. There are two main print-head shapes for producing the dots. For one of these the print head consists of a 5 by 7 or 7 by 9 matrix of tiny heating elements. To print a character the head is moved to a character position and the dot-sized heating elements for the desired character are turned on. After a short time the heating elements are turned off and the head is moved to the next character position. Printing then is done one complete character at a time.

The second print-head configuration for thermal dot-matrix printers has the heating elements along a metal bar which extends across the entire width of the paper.

There is a heating element for each dot position on a print line, so this type can print an entire line of dots at a time. The metal bar removes excess heat. Characters and graphics are printed by stepping the paper through the printer one dot line at a time. A few thermal printers can print up to 400 lines/min.

Some of the newer thermal printers have the heat-sensitive material on a ribbon instead of on the paper. When a spot on the ribbon is heated, a dot of ink is transferred to the paper. This approach makes it possible to use standard paper and, by switching ribbons, to print color graphics as well as text.

The main advantage of thermal printers is their low noise. Their main disadvantages are: the special paper or ribbon is expensive, printing carbon copies is not possible, and most thermal printers with good print quality are slow.

Laser and Other Page Printers

These printers operate on the same principle as most office copiers. The basic approach is to first form an image of the page that is to be printed on a photosensitive drum in the machine. Powdered ink, or "toner," is then applied to the image on the drum. Next the image is electrostatically transferred from the drum to a sheet of paper. Finally the inked image on the paper is "fused" with heat.

In an office copy machine a camera lens is used to produce an image of the original on the photo-sensitive drum. In page printers used with computers, there are three common methods of producing the image on the drum. The most common methou and the one that gives us the name laser printer is with a laser, as shown in Figure 13-44. A rotating mirror sweeps a laser beam across the photosensitive drum as it rotates. The laser beam is turned on and off as it is swept back and forth across the drum to produce an image in about the same way that an image is produced on a raster scan CRT. After the image on the drum is inked and transferred to the paper, the drum is cleaned and is ready for the next page.

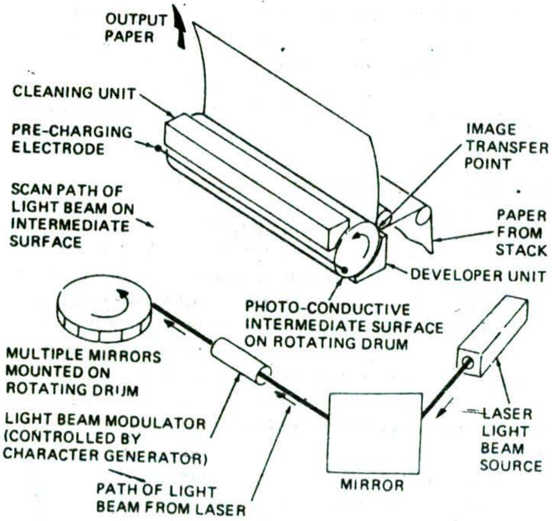

OUTPUT PAPER
CLEANING UNIT
PRE-CHARGING ELECTRODE
IMAGE TRANSFER POINT
SCAN PATH OF LIGHT BEAM ON INTERMEDIATE SURFACE
PAPER FROM STACK
DEVELOPER UNIT
PHOTO-CONDUCTIVE INTERMEDIATE SURFACE ON ROTATING DRUM
MULTIPLE MIRRORS MOUNTED ON ROTATING DRUM
LIGHT BEAM MODULATOR (CONTROLLED BY CHARACTER GENERATOR)
LASER LIGHT BEAM SOURCE
MIRROR
PATH OF LIGHT BEAM FROM LASER

FIGURE 13-44 Laser printer mechanism. *(Courtesy DATAPRODUCTS Corporation.)*

A second way of producing the dots on the photosensitive drum is with a linear array of tiny LEDs. The image is generated one line at a time as the drum rotates. This approach has less moving parts than the laser approach, but if an LED burns out, it will leave a blank streak through the printout.

The third common method of producing the image on the drum is with a linear array of tiny liquid-crystal "shutters." When the shutter is opened, the light from a bright backlight exposes a spot on the drum. As with the other methods, the image is produced on the drum one line at a time.

One major advantage of laser and other page printers is their high print quality. Commonly available lower-priced units have a resolution of 300 dots per inch, and the next generation will probably extend this to 600 dots per inch. For comparison, 1200 to 2400 dots per inch are commonly used for high-quality typesetting. Print speeds are in the range of 10 to 12 pages per minute for text and 1 to 4 pages per minute for graphics.

The control circuitry in, for example, a laser page printer is much more complex than that in an impact-type dot-matrix printer because the image is developed as a large dot matrix with many dots. To generate complex images as rapidly as possible, these printers often use a high-speed microprocessor with a graphics processor, similar to the CRT system shown in Figure 13-23. The dot patterns for several different character fonts are usually included in the ROMs in the controller so that you are not restricted to just one character set. Some of these printers also allow you to download custom character fonts to RAM in the printer. Several megabytes of RAM are needed in the printer to hold the data for complex graphics images.

To produce graphics and page layouts you write a program using a *printer control language* or *PCL*. The three most common languages are HP's PCL-4, Cannon's CaPCL, and Adobe's Postscript. These languages allow you to specify where to draw lines on the page, the scale factor for characters, gray shading, and many other page features.

Ink-Jet Printers

Still another type of printer that uses a dot-matrix approach to produce text and graphics is the ink-jet. Early ink-jet printers used a pump and a tiny nozzle to send out a continuous stream of tiny ink globules. These ink globules were passed through an electric field, which left them with an electrical charge. The stream of charged ink globules was then electrostatically deflected to produce characters on the paper in the same way that the electron beam is deflected to produce an image on a CRT screen. Excess ink was deflected to a gutter and returned to the ink reservoir. Ink-jet printers are relatively quiet, and some of these electrostatically deflected ink-jet printers can print up to 45,000 lines/min. Several disadvantages, however, prevented them from being used more widely. They tend to be messy and difficult to keep working well. Print quality at high speeds is poor and multiple copies are not possible.

Newer ink-jet printers use a variety of approaches to solve these problems. Some, such as the HP Thinkjet, use ink cartridges which contain a column of tiny heaters. When one of these tiny heaters is pulsed on, it causes a drop of ink to explode onto the paper. Others, such as the IBM Quietwriter, for example, use an electric current to explode microscopic ink bubbles from a special ribbon directly onto the paper. These last two approaches are really hybrids of thermal and ink-jet technologies. They can produce very near letter-quality print at speeds comparable to those of slower dot-matrix impact printers. A disadvantage of some ink-jet printers is that they require special paper for best results.

SPEECH SYNTHESIS AND RECOGNITION WITH A COMPUTER

In a great many cases it is very convenient for a computer to communicate verbally with a user. Some examples of the use of computer-created speech are talking games, talking cash registers, and text-to-speech machines used by blind people. Other examples are medical monitor systems that give verbal warnings and directions when some emergency condition exists. This use demonstrates some of the major advantages of speech readout. The verbal signal attracts more attention than a simple alarm, and the user does not have to search through a series of readouts to determine the problem.

Adding speech recognition circuitry to a computer so that it can interpret verbal commands from a user also makes the computer much easier to use. The pilot of a rocket ship or space shuttle, for example, can operate some controls verbally while operating other controls manually. (It probably won't be too long before we eliminate the verbal/manual link and control the whole ship directly from the brain, but that is another story, perhaps in the next book.) Voice entry systems are also useful for handicapped programmers and other computer users. We will first describe for you the different methods used to create speech with a computer and then describe some speech-recognition methods.

Speech-Synthesis Methods

There are several common methods of producing speech from a computer. The trade-offs between the different methods are speech quality and the number of bits that must be stored for each word. In other words, the higher the speech quality you want, the more bits you have to store in memory to represent each word and the faster you have to send bits to the synthesizer circuitry. All the common methods of speech synthesis fall into two general categories: waveform modification and direct digitization. In order to explain how the waveform-modification approaches work, we need to talk briefly about how humans produce sounds.

VAVEFORM-MODIFICATION SPEECH SYNTHESIS

Some speech sounds, called voiced sounds, are produced by vibration of the vocal cords as air passes from the lungs. The frequency of vibration or *pitch*, the position of the tongue, the shape of the mouth, and the position of the lips determine the actual sound produced. The vowels A and E are examples of voiced sounds. Another type of sound, called unvoiced sound, is produced by modifying the position of the tongue and the shape of the mouth as a constant stream of air comes from the lungs. The letter S is an example of this type of sound. A third type of sound, the nasal sounds—called *fricatives*—consist of a mixture of voiced and unvoiced sounds. In electronic terms then, the human vocal system consists of a variable-frequency signal generator as the source for voiced sounds, a "white" noise signal source for unvoiced sounds, and a series of filters which modify the outputs from the two signal sources to produce the desired sounds. Figure 13-45 shows this in block diagram form.

The three main approaches to implementing this model electronically are *linear predictive coding* or LPC, *formant filtering*, and *phoneme synthesis*. These methods differ mostly in the type of filter used and in how often the filter characteristics are updated.

LPC synthesizers, such as that in the Texas Instruments "Speak and Spell," use a digital filter such as we described in Chapter 10 to modify the signals from a pulse and a white noise source. For this type of filter the parameters that must be sent from the microcomputer are the coefficients for the filter and the pitch for the pulse source. Remember from the discussion in Chapter 10 that for a digital filter, the current output value is computed or "predicted" as the sum of the current input value and portions of previous input values. A high-quality LPC synthesizer may require as many as 16 Kbits/s. An example of a currently available LPC speech chip is the TI TSP50C10. For further information about LPC synthesis, consult the data sheet for this device.

The formant filter speech-synthesis approach uses several resonant or *formant* filters to massage the signals from a variable-frequency signal source and a white noise source. Figure 13-46, page 482, shows how the frequencies of these formant filters might be arranged for a male and for a female voice. For this type of system the parameters that must be sent from the computer are the pitch of the variable-frequency signal, the center frequency for each formant filter, and the bandwidth of each formant filter. The data rate for direct formant

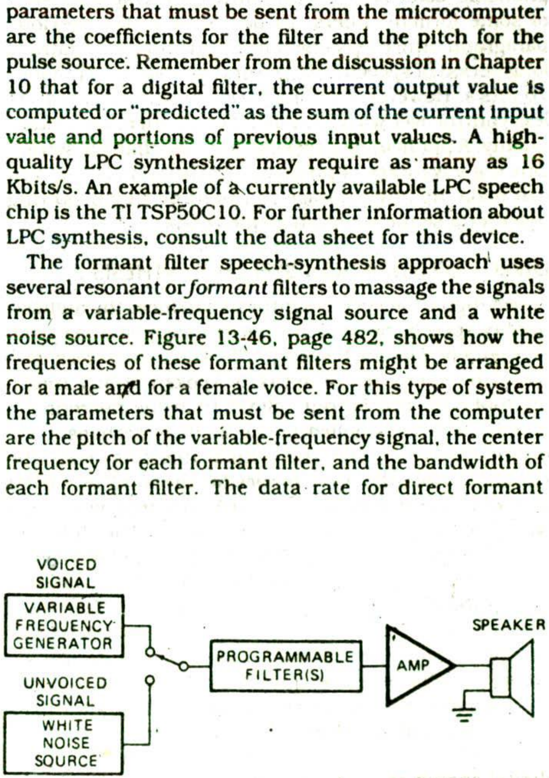


FIGURE 13-45 Electronic model of human vocal tract.

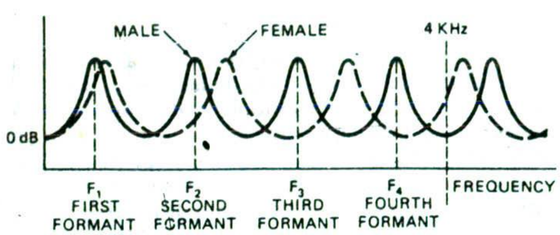


FIGURE 13-46 Filter responses for formant speech synthesizer.

synthesis is only about 1 Kbit/s, but the parameters must be determined with complex equipment. It is not easy to develop a custom vocabulary for a specific application. A phoneme approach solves this problem and requires a still lower data rate at the expense of lower speech quality.

Phonemes are fragments of words. An example of a phoneme speech synthesizer is the Artic Technologies 263A, which can be interfaced with a microcomputer port or in some cases interfaced directly to microprocessor buses. Words are produced by sending a series of 6-bit phoneme codes to the device. Five internal 8-bit registers also allow you to control parameters such as speech rate, pitch, amplitude, articulation rate, and vocal tract filter response. Inside the 263A the 6-bit phoneme code is used to control the characteristics of some formant filters, as described in the previous paragraph. Since only one code is sent out for a relatively long period of speech, the required bit rate is only about 70 bits/s. However, the long period between codes gives less control over waveform details and, therefore, sound quality. A phoneme synthesizer has a mechanical sound. One big advantage of phoneme synthesizers is that you can make up any message you want by simply putting together a sequence of phoneme codes.

DIRECT DIGITIZATION SPEECH SYNTHESIS

Direct digitation speech synthesis produces the highest-quality speech, because it is essentially just a playback of digitally recorded speech. To start, the word you want the computer to speak is spoken clearly into a microphone. The output voltage from the microphone is amplified and applied to the input of perhaps a 12-bit A/D converter. One approach at this point might be to simply store the A/D samples for the word in a ROM and read the values out to a D/A converter when you want the computer to speak the word. The difficulty with this approach is that if the samples are taken often enough to produce good speech quality, a lot of memory is required to store the samples for a word. To reduce the amount of memory required, several speech-compression algorithms are used. These algorithms are too complex to discuss here, but the basic principles involve storing repeated waveforms only once, taking advantage of symmetry in waveforms, and not storing values for silent periods. Even with compression, however, direct digital speech requires considerable memory and a bit rate as high as 64 Kbits/s. To further reduce the memory required for direct digital speech, some systems use differential or *delta* modulation. In these systems only a 3-bit or 4-bit code, representing how much a sample has changed from the last sample, is stored in memory instead of storing the complete 12-bit value. This system works well for audio signals, since they change slowly.

The OKI Semiconductor MSM6388 device contains much of the circuitry needed to digitize and reproduce speech using adaptive-differential pulse code modulation (ADPCM). This device contains a microphone preamplifier, A/D converter, D/A converter, and some low-pass filters. The digital values produced by the A/D converter are stored in external memory. About 260 s of speech can be stored in 4 Mbits of external memory.

Another example of a direct digital synthesis system is the National Semiconductor *Digitalker*. For further information, consult the data sheets for these devices.

Speech Recognition

Speech recognition is considerably more difficult than speech synthesis. The process is similar to trying to recognize human faces with a computer vision system. With most speech-recognition systems the first step is to train the system, or, in other words, produce templates for each of the words that the system needs to recognize and store these templates in memory. To produce a template for a word, the intended user speaks the word several times into a microphone connected to the system. The system then determines several parameters or *features* for each repetition of the word and averages them to produce the actual template.

Different systems extract different parameters to form the template. One of the most common methods uses a set of formant filters with their center frequencies adjusted to match those of the average speaker. The output amplitude of each formant filter is averaged to produce a signal proportional to the energy in that frequency band. Also used are one or more zero-crossing detectors to give basic frequency information. The pulse train from the zero-crossing detector is converted to a proportional voltage, so it can be digitized along with the outputs from the formant averagers.

When a word is spoken, samples of each of the features are taken and digitized at evenly spaced intervals of 10 to 20 ms during the duration of the word. The features are stored in memory. If this is a training run, the set of samples will be averaged with others to form the template for the word. If this is a recognition run, this set of features will be compared with the templates stored in memory. The best match is assumed to be the correct word. Currently none of the available voice-recognition systems is 100 percent accurate, but they are improving.

The best current example of speech recognition is probably the Dragon Systems Inc. DragonDictate which consists of a PC- or PS/2-compatible plug-in board and software. This system has a built-in vocabulary of 25,000 words and allows the user to define up to 5000 more words. This unit is intended for use in speech-to-text applications such as generating reports. When a user speaks a word, the system looks up the most likely match and sends the match word to the screen. If the word is incorrect, the user can correct the word verbally or with the keyboard. The system is adaptive, so its recognition rate improves with continued use. Incidentally, the DragonDictate system uses a TMS32010 digital signal processor device we described in Chapter 10 to filter the input signal.

A less expensive PC-compatible speech-recognition unit is the VPC 2000 from VOTAN Inc. In addition to recognizing words or phrases, this unit also has a built-in voice-activated telephone dialing and answering service. Another PC-compatible unit, the VocaLink from Interstate Voice Products, permits the programming of up to 240 spoken commands to control standard PC software such as word processors and business programs. Perhaps the HAL 9000 is not too far away.

DIGITAL VIDEO INTERACTIVE

In this chapter we have shown you how text and graphics images are produced on a CRT; how text, speech, and graphics data are stored on magnetic or optical disks; and how a computer can be used to recognize and generate speech. To us the most exciting applications of all these technologies are compact digital-interactive (CD-I) and digital video interactive (DVI). CD-I was developed by Phillips and DVI was developed by the David Sarnoff Research Center and later purchased by Intel Corporation, which is continuing its development. The systems are very similar, but we will concentrate on DVI for this discussion.

The Intel DVI system consists of some very powerful software and two circuit boards which plug into a PC- or PS/2-type computer. This system allows up to 72 min of full-motion video images and stereo sound to be produced on the computer from a single 5¼-in. optical disk. There are two very significant points about this.

First, the system is interactive. This means that the image and sound output at any particular time depends on the input that you supply with the keyboard, a mouse, or perhaps a joystick. One example called Design and Decorate, which was developed to demonstrate DVI, allows you to place furniture in a room, move the furniture around in the room, and even reupholster the furniture with different fabrics. Another DVI demonstration allows you to fly a plane around "real" landscapes. Another demo allows you to landscape an image of your house and see how it will look as the plants mature. Still another demo teaches you how to use a camera. This demo allows you to focus the camera and see the effect of f-stop on depth of field, etc. Perhaps you can see from these brief discussions that DVI has great potential for individualized education, entertainment, and marketing.

The technique that makes DVI possible is audio and video compression. As we explained in a previous section on digitizing speech signals, adaptive-differential pulse code modulation can be used to reduce the amount of data required to store a digitized audio signal. This technique is based on the fact that audio signals change relatively slowly, so instead of storing a value for the entire amplitude at each point on the signal, only a value for the change from the previous data point is stored.

Most of the time video images also change relatively slowly. The amount of data required to store a sequence of video images can be drastically reduced by storing the data for the first frame in the sequence and then just storing changes from that frame for the rest of the frames in the sequence. Further reduction can be accomplished by taking advantage of the fact that the resolution of the human eye is not as fine for color images as it is for monochrome images. The DVI system stores the color data for every fourth pixel and interpolates to get the color values for the pixels in between these.

To give you an idea of how important this video compression is, remember from the discussions earlier in the chapter that about 153 Kbytes of memory are required to store the pixel data for one frame of a 640 × 480 × 16 color display. With a refresh rate of 60 frames/s, a 648-Mbyte optical disk could hold only about 70 s of video frames. The DVI system requires an average of only about 5 Kbytes to store the data for a frame.

The steps involved in developing an application using the DVI system are as follows:

1. Use the edit level video (ELV) editor that comes with the system to digitize the basic video image sequence and reduce the resolution of the images to 256 × 240 pixels.
2. Digitize the audio signal.
3. Use ELV editor to select the desired video and audio sequences.
4. Add text, graphics images, and control programming.
5. Send the original video and the editor output to Intel or some other vendor who will use a high-speed parallel computer to produce the compressed video and audio data. This output is called presentation level video (PLV).
6. Combine the PLV with the ELV editor output and use the result to program a WORM or EO disk.

For further information on developing DVI applications, see the references in the Bibliography.

Once the DVI disk is programmed, it can be "played" on a compatible optical disk drive and run with the DVI software. The DVI boards decompress the video image data, decompress the audio and data, and interact with the user. Perhaps in the not-too-distant future, systems such as this will let you pilot the "Enterprise" through an adventure of your own.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

Extended ASCII codes

CRT operation
- Raster scan display
- Field
- Interlaced and noninterlaced scanning
- Frame rate
- Video monitor
- CRT terminal
- Horizontal and vertical sync pulses
- Composite video
- Character generator
- Display-refresh RAM or frame buffer
- Dot clock
- Overscan
- Attribute code
- Bit-mapped graphics display
- Picture element, pixel, or pel
- Packed pixel storage
- Planar pixel storage
- Palette DAC, RAMDAC
- VRAM
- CGA, EGA, VGA, 8514/A display adapters
- Bresenham's algorithm
- BITBLT
- Graphics processor, RISC processor, CISC processor
- Megaflops, Mflops
- TIGA

Liquid Crystal computer displays

Plasma computer displays

Computer mouse, trackball

Computer vision
- Video cameras—vidicon
- CCD cameras

Floppy disks and hard disks
- FM, MFM, and RLL encoding
- Access time
- Hard and soft sectoring
- Index holes
- Index, ID, and data fields, gaps, address marks
- Cyclic redundancy character
- Cylinders
- ST-506, ESDI, and SCSI disk interfaces
- Low-level format, high-level format
- Cluster, file allocation table, directory
- DOS function calls—how called
- File-control block, file handle
- Open and close a file
- C FILE pointer data type
- C buffered I/O and streams
- RAM disk
- Disk cache

Optical disk systems
- OROM, WORM, EO type systems

Printer mechanisms
- DOT-matrix impact and Dot-matrix thermal
- Laser and other page printers
- Ink jet

Speech synthesis
- Pitch, unvoiced sounds, and fricatives
- Linear predictive coding, formant, phoneme
- Direct digitization

Speech recognition

Digital video interactive

REVIEW QUESTIONS AND PROBLEMS

1. *a.* Why are the predefined functions such as scanf and getche not always suitable for reading keycodes from a PC- or PS/2-type computer?
   b. Use the program in Figure 13-3 to help you write a C program section which calls the BIOS INT 16H procedure to wait until a key is pressed, and then returns the code for the pressed key (assume only standard ASCII).

2. With the help of a simple drawing, explain how a noninterlaced raster is produced on a CRT.

3. Use a simple drawing to help you describe how a display of the letter X is produced by the electron beam on a noninterlaced raster-scan CRT display.

4. Refer to Figure 13-6 to help you answer the following questions.
   a. What is the purpose of the RAM in this circuit?
   b. At what point(s) in displaying a frame do the address inputs of this RAM get changed?
   c. At what point(s) in displaying a frame do the R0-R3 address inputs of the character-generator ROM get changed?
   d. What is the purpose of the shift register on the output of the character generator ROM?
   e. At what point(s) in displaying a frame are horizontal sync pulses produced?
   f. At what point(s) in displaying a frame are vertical sync pulses produced?

5. A CRT display is designed to display 24 character rows with 80 characters in each row. The system uses a 7 by 9 character generator in a 9 by 12 dot matrix. Assuming a 60-Hz noninterlaced frame rate, three additional character times for horizontal overscan, and 120 additional scan lines for vertical overscan, find the following values.
   a. Total number of character times/row
   b. Total number of scan lines/frame
   c. Horizontal frequency (number of lines/second)
   d. Dot-clock frequency (dots/second)
   e. Minimum bandwidth required for video amplifier
   f. Time between RAM accesses

6. The IBM PC color adapter board uses a 14-MHz dot

clock frequency, a 15.750-kHz horizontal scan rate, and a 60-Hz frame rate. Characters are produced in an 8 by 8 dot matrix. There are 80 characters/row and 25 rows/frame.

a. What is the total number of dot times per scan line?

b. How many dot times then are left for horizontal overscan?

c. What is the total number of scan lines per frame including overscan?

d. How many scan lines then are left for vertical overscan?

7. How does the CRT display system in Figure 13-5 arbitrate the dispute that occurs when the 6845 CRT controller and the microprocessor both want to access the display RAM at the same time?

8. Write a program which uses the IBM BIOS procedures to read a string of characters entered from the keyboard, put the key codes in a buffer in memory, and display the characters for the pressed keys on the CRT.

9. How much memory is required to store the pel data for a bit-mapped monochrome 640 by 480 display?

10. Describe how three electron beams are used to produce all possible colors on a color CRT screen.

11. *a.* How many memory bits are required to store the data for a pixel that can be any one of 256 colors?

b. How much memory is required to store the pel data for a 1024 by 768 display where each pel can be any one of 16 colors?

c. Use diagrams to help you explain the difference between packed pixel storage and planar pixel storage.

12. Use a diagram to help you explain how an EGA system uses palette registers to produce a display of 16 colors from a palette of 64 colors.

13. Mode 13H of a VGA system produces a display of 256 colors from a palette of 256K possible colors.

a. How many bits are required to specify one of 256K colors?

b. Why is it currently impractical to store the pixel data for a direct display of 256K colors?

c. Draw a diagram showing how the VGA color registers and palette registers are used to specify one of 256 colors out of a palette of 256K.

d. Describe how the actual red, green, and blue drive signals are produced from the color register values in a VGA system.

14. *a.* Write assembly language instructions which use the BIOS INT 10H procedure to initialize a VGA adapter for 320 × 200 × 256 color mode.

b. Add instructions which position the cursor approximately in the center of the screen and write your name at that location.

15. Explain the purpose of the following statements or groups of statements in the C graphics program in Figure 13-23.

a. Initgraph(&driver,&mode,"c:\\tc\\BGI");

b. window_size = imagesize(160,100,480 ,250); window_buffer = malloc (window_size);

c. getimage(160,100,480,250);

d. putimage(160,100,window_buffer,0);

e. free(window_buffer)

16. *a.* Why do many microcomputers now use a dedicated graphics processor instead of having the main processor compute pixel values for graphics images?

b. Why is a math coprocessor often included in the design of a graphics processor system?

17. What are the major advantages of LCD displays over CRT displays for use in portable microcomputers?

18. The vector graphics approach is an alternative to the raster scan approach of producing graphics displays on a CRT screen. In a vector graphics display system the beam is directly moved from point to point on the CRT screen to trace out images. The most common way to direct the beam is by connecting a D/A converter to the X axis drive and another D/A converter to the Y axis drive. For this problem, assume the inputs of an 8-bit D/A converter are connected to port FFF8H of a microcomputer and the output of the D/A converter is connected to the X axis of an oscilloscope. The inputs of another 8-bit D/A converter are connected to port FFFAH of a microcomputer, and the output of this D/A is connected to the Y axis of the oscilloscope. Write a program which uses these D/A converters to display a square on the screen of the oscilloscope. Then modify the program so that the square enlarges after each 100 refreshes.

19. Describe how a CCD camera produces pixel data which can be stored in computer memory.

20. *a.* Describe the mechanical mechanisms used to move the read/write head to the desired track on a floppy or hard disk.

b. Explain why a magnetic hard disk can store much more data than a floppy disk and why data can be read from a hard disk much faster than it can from a floppy.

c. Explain why a phase-locked loop is used as part of the interface circuitry for a floppy or hard disk drive.

d. What is the major advantage of RLL 2,7 encoding over MFM encoding for recording data on magnetic disks?

21. *a.* What is the main improvement of an ESDI hard disk interface over the older ST-506 interface?

b. Draw a diagram to help explain how an SCSI I/O bus is connected in a system and how it operates.

22. *a.* Describe the purpose of the CRC bytes included with each block of data recorded on the disk.

b. Describe how you format a floppy disk on a DOS-based system.

c. Describe the three steps involved in formatting and partitioning a blank magnetic hard disk.

23. *a.* Describe the purpose of the file allocation table written on a disk by DOS.

b. If a data file requires several clusters on a disk, how does a DOS keep track of where the pieces of the file are located?

c. List the major types of information contained in the directory entry for each file in a DOS system.

24. Write a program which uses the IBM PC DOS function calls to read in a string containing your name from the keyboard to a buffer in memory and sends the string to a printer. Remember to use the DOS 4CH function call to return to DOS at the end of the program.

25. Write a program which uses DOS function calls to read a line of text from the keyboard to a buffer in memory and then, when the carriage return key is pressed, opens a file and writes the text to the file.

26. Explain the operation performed in Figure 13-42 by each of the following C statements or group of statements more thoroughly than the comments.

a. FILE *fp;

b.
```
if((fp=open(filename,"wt")) = =0)
{
perror(filename)
}
```

c. fclose();

d.
```
while(!feof(fp))
fputc(fgetc(fp),stdout);
```

27. *a.* Describe the operation of a RAM disk and explain how it speeds up the execution of some programs.

b. Explain the operation of a disk cache and explain how it is different from a RAM disk.

28. *a.* Describe how stored data is read from optical disks and describe the advantages this readout method has over that used for hard magnetic disks.

b. List the major advantages of optical disk data storage over magnetic hard disk data storage.

c. Describe how data bits are recorded in magneto-optic erasable optical (EO) disk systems.

29. A human brain can store about 10^{10} bits of data and has an access time in the order of about a second. Compare these parameters with those of an optical disk system such as the Maxtor Tahiti I discussed in the text.

30. Describe the operation of the print mechanism for each of the following types of printer. Also give an advantage and a disadvantage for each type.

a. Impact-type dot-matrix

b. Thermal

c. Laser

d. Ink-jet

31. What are the major differences between an LPC speech synthesizer and a formant speech synthesizer?

32. Describe the operation of a direct-digitization speech synthesizer. As part of the description give the major advantage and the major disadvantage of this type speech synthesis.

33. *a.* Digital video interactive systems allow up to 72 min of full motion video and sound to be recorded on a single 5-in. optical disk. Describe the techniques used to store this immense amount of data on the disk.

b. Describe how a DVI system might be used to teach you how to fly a space shuttle.

CHAPTER 14

Data Communication and Networks

In Chapter 2 we discussed "computerizing" an electronics factory. What this means is that computers are integrated into all the operations of the factory and that each person in the company has access to a computer. The company may have a large centrally located mainframe computer, several minicomputers that serve groups of users, individual computer engineering workstations, and portable computers spread around the world with its salespeople. In order for all these computers to work together, they must be able to communicate with each other in an organized manner. In this chapter we show you some of the devices, signal standards, and systems used for communication with and between computers.

In the first section of the chapter we discuss the hardware and low-level software required to interface microcomputer buses to serial data communication lines. Then we discuss how the serial data signals are transmitted from one place to another. This discussion includes RS-232C-type standards, modems, and fiber-optic cables. The next section of the chapter shows you how to write programs which perform simple serial data communication. As an example in this section we use a program which allows you to download programs from a PC- or PS/2-type computer to an SDK-86 board. In the final sections of the chapter we discuss the operation of several common computer networks.

OBJECTIVES

At the end of this chapter, you should be able to:

1. Show and describe the meaning of the bits in the format used for sending asynchronous serial data.
2. Initialize a common UART for transmitting serial data in a specified format.
3. Describe several voltage, current, and light (fiber-optic) signal methods used to transmit serial data.
4. Describe the function of the major signals in the RS-232C standard.
5. Show how to connect RS-232C equipment directly or with a "null-modem" connection.
6. Describe the different types of modulation commonly used by modems.
7. Use the IBM PC BIOS, DOS, and C procedures to send and receive serial data.
8. Show the formats for a byte-oriented protocol and for a bit-oriented protocol used in synchronous serial data transmission.
9. Draw diagrams to show the common computer network topologies.
10. Describe the operation of an Ethernet system.
11. Describe the operation of a token-passing ring system.
12. Show the major signal groups for the GPIB (IEEE 488) bus, describe how bus control is managed, and describe how data is transferred on a handshake basis for the GPIB.

INTRODUCTION TO ASYNCHRONOUS SERIAL DATA COMMUNICATION

Overview

Serial data communication is a somewhat difficult subject to approach, because you need pieces of information from several different topics in order for each part of the subject to really make sense. To make this approach easier, we will first give an overview of how all the pieces fit together and then describe the details of each piece later in specific sections. A problem with this subject is that it contains a great many terms and acronyms. To help you absorb all of these, you may want to make a glossary of terms as you work your way through the chapter.

Within a microcomputer data is transferred in parallel, because that is the fastest way to do it. For transferring data over long distances, however, parallel data transmission requires too many wires. Therefore, data to be sent long distances is usually converted from parallel form to serial form so that it can be sent on a single wire or pair of wires. Serial data received from a distant source is converted to parallel form so that it can easily be transferred on the microcomputer buses. Three terms often encountered in literature on serial data systems are *simplex*, *half-duplex*, and *full-duplex*. A simplex data line can transmit data only in one direction. An earthquake sensor sending data back from Mount St.

Helens or a commercial radio station, are examples of simplex transmission. Half-duplex transmission means that communication can take place in either direction between two systems, but can only occur in one direction at a time. An example of half-duplex transmission is a two-way radio system, where one user always listens while the other talks because the receiver circuitry is turned off during transmit. The term full-duplex means that each system can send and receive data at the same time. A normal phone conversation is an example of a full-duplex operation.

Serial data can be sent *synchronously* or *asynchronously*. For synchronous transmission, data is sent in blocks at a constant rate. The start and end of a block are identified with specific bytes or bit patterns. In a later section of the chapter we discuss synchronous data transmission in detail. For asynchronous transmission, each data character has a bit which identifies its start and 1 or 2 bits which identify its end. Since each character is individually identified, characters can be sent at any time (asynchronously), in the same way that a person types on a keyboard.

Figure 14-1 shows the bit format often used for transmitting asynchronous serial data. When no data is being sent, the signal line is in a constant high or *marking* state. The beginning of a data character is indicated by the line going low for 1 bit time. This bit is called a *start* bit. The data bits are then sent out on the line one after the other. Note that the least significant bit is sent out first. Depending on the system, the data word may consist of 5, 6, 7, or 8 bits. Following the data bits is a parity bit, which—as we explained in Chapter 11—is used to check for errors in received data. Some systems do not insert or look for a parity bit. After the data bits and the parity bit, the signal line is returned high for at least 1 bit time to identify the end of the character. This always-high bit is referred to as a *stop bit*. Some older systems use 2 stop bits. For future reference note that the efficiency of this format is low, because 10 or 11 bit times are required to transmit a 7-bit data word such as an ASCII character.

The term *baud rate* is used to indicate the rate at which serial data is being transferred. Baud rate is defined as 1/(the time between signal transitions). If the signal is changing every 3.33 ms, for example, the baud rate is 1/(3.33 ms), or 300 Bd. There is an almost unavoidable, but incorrect, tendency to refer to this as 300 bits/s. In some cases, the two do correspond, but in other cases 2 or more actual data bits are encoded in one signal transition, so data bits per second and baud do not correspond. Common baud rates are 300, 600, 1200, 2400, 4800, 9600, and 19,200.

To interface a microcomputer with serial data lines, the data must be converted to and from serial form. A parallel-in-serial-out shift register and a serial-in-parallel-out shift register can be used to do this. Also needed for some cases of serial data transfer is handshaking circuitry to make sure that a transmitter does not send data faster than it can be read in by the receiving system. There are available several programmable LSI devices which contain most of the circuitry needed for serial communication. A device such as the National INS8250, which can only do asynchronous communication, is often referred to as a *universal asynchronous receiver-transmitter* or UART. A device such as the Intel 8251A, which can be programmed to do either asynchronous or synchronous communication, is often called a *universal synchronous-asynchronous receiver-transmitter* or USART.

Once the data is converted to serial form, it must in some way be sent from the transmitting UART to the receiving UART. There are several ways in which serial data is commonly sent. One method is to use a current to represent a 1 in the signal line and no current to represent a 0. We discuss this *current-loop* approach in a later section. Another approach is to add line drivers on the output of the UART to produce a sturdy voltage signal. The range of each of these methods, however, is limited to a few thousand feet.

For sending serial data over long distances, the standard telephone system is a convenient path, because the wiring and connections are already in place. Standard phone lines, often referred to as *switched lines* because any two points can be connected together through a series of switches, have a bandwidth of only about 300 to 3000 Hz. Therefore, for several reasons, digital signals of the form shown in Figure 14-1 cannot be sent directly over standard phone lines.

NOTE: Phone lines capable of carrying digital data directly can be leased, but these are somewhat costly and are limited to the specific destination of the line.

The solution to this problem is to convert the digital signals to audio-frequency tones, which are in the frequency range that the phone lines can transmit. The device used to do this conversion and to convert transmitted tones back to digital information is called a *modem*. The term is a contraction of modulator-

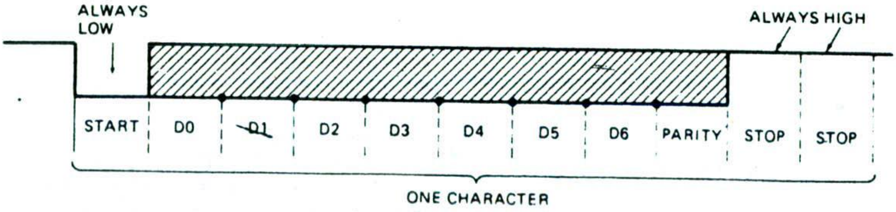
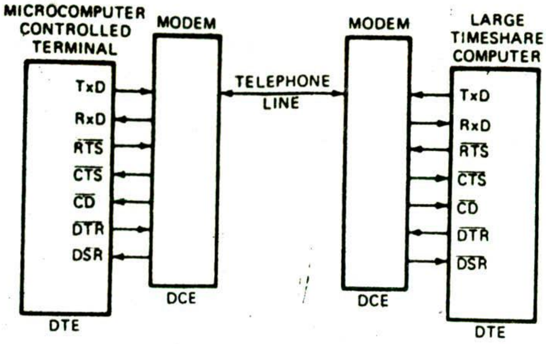


FIGURE 14-1 Bit format used for sending asynchronous serial data.



DTE = DATA TERMINAL EQUIPMENT
DCE = DATA COMMUNICATION EQUIPMENT

FIGURE 14-2 Digital data transmission using modems and standard phone lines.

demodulator. In a later section of this chapter we discuss the operation of some common types of modems. For now, take a look at Figure 14-2, which shows how two modems can be connected to allow a remote terminal to communicate with a distant mainframe computer over a phone line. Modems and other equipment used to send serial data over long distances are known as *data communication equipment* or DCE. The terminals and computers that are sending or receiving the serial data are referred to as *data terminal equipment* or DTE.

The data and handshake signal names shown in Figure 14-2 are part of a serial data communications standard called RS-232C, which we discuss in detail in a later section. For now you just need enough of an overview of these signals so that the initialization of the 8251A UART in the next section makes sense to you. Note the direction arrowheads on each of these signals. Here is a sequence of signals that might occur when a user at a terminal wants to send some data to the computer.

After the terminal power is turned on and the terminal runs any self-checks, it asserts the *data-terminal-ready* ($\overline{\text{DTR}}$) signal to tell the modem it is ready. When it is powered up and ready to transmit or receive data, the modem will assert the *data-set-ready* ($\overline{\text{DSR}}$) signal to the terminal. Under manual control or terminal control, the modem then dials up the computer.

If the computer is available, it will send back a specified tone. Now, when the terminal has a character actually ready to send, it will assert a *request-to-send* ($\overline{\text{RTS}}$) signal to the modem. The modem will then assert its *carrier-detect* ($\overline{\text{CD}}$) signal to the terminal to indicate that it has established contact with the computer. When the modem is fully ready to transmit data, it asserts the *clear-to-send* ($\overline{\text{CTS}}$) signal back to the terminal. The terminal then sends serial data characters to the modem. When the terminal has sent all the characters it needs to, it makes its $\overline{\text{RTS}}$ signal high. This causes the modem to unassert its $\overline{\text{CTS}}$ signal and stop transmitting. A similar handshake occurs between the modem and the computer at the other end of the data link. The important point at this time is that a set of handshake signals is defined for transferring serial data to and from a modem.

Now that you have an overview of asynchronous serial data, modems, and handshaking, we will describe the operation of a device commonly used to interface a microcomputer to a modem or other device which requires serial data.

An Example USART—The Intel 8251A

SYSTEM CONNECTIONS AND SIGNALS

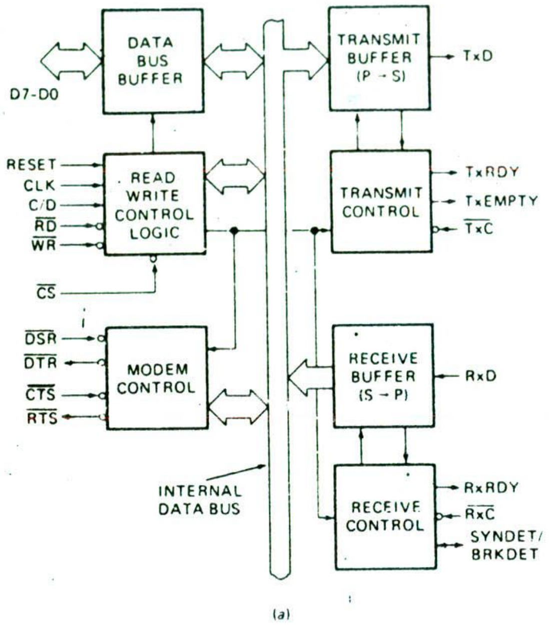
As we showed you in Chapter 7, an 8251A is used as the serial port on SDK-86 boards. It is also used on the IBM PC synchronous communication board and on many other boards, so we chose to use it as an example here.

Figure 14-3, page 490, shows a block diagram and the pin descriptions for the 8251A, and Figure 7-6, sheet 9, shows how an 8251A is connected on the SDK-86 board. Keep copies of these two handy as you work your way through the following discussion.

As shown in the SDK-86 schematic, the eight parallel lines, D7–D0, connect to the system data bus so that data words and control/status words can be transferred to and from the device. The *chip select* ($\overline{\text{CS}}$) input is connected to an address decoder so the device is enabled when addressed. The 8251A has two internal addresses, a control address, which is selected when the C/$\overline{\text{D}}$ input is high, and a data address, which is selected when the C/$\overline{\text{D}}$ input is low. For the SDK-86 the control/status address is FFF2H and the data read/write address is FFF0H. The RESET, $\overline{\text{RD}}$, and $\overline{\text{WR}}$ lines are connected to the system signals with the same names. The clock input of the 8251A is usually connected to a signal derived from the system clock to synchronize the internal operations of the USART with the processor timing. In the case of the SDK-86 the clock input is connected to the 2.45-MHz PCLK signal, which is derived from the processor clock but has a frequency the 8251A can handle.

The signal labeled TxD on the upper right corner of the 8251A block diagram is the actual *serial-data* output. The pin labeled RxD is the *serial-data* input. The additional circuitry connected to the TxD pin on the SDK-86 board is needed to convert the TTL logic levels from the 8251A to current loop or RS-232C signals. The circuitry connected to the RxD pin performs the opposite conversion. We will discuss current loop and RS-232C signal standards a little later.

The shift registers in the USART require clocks to shift the serial data in and out. $\overline{\text{TxC}}$ is the *transmit shift-register clock* input, and $\overline{\text{RxC}}$ is the *receive shift-register clock* input. Usually these two inputs are tied together so they are driven by the same signal. If you look at Figure 7-6, sheet 9, you should see how some wire-wrap jumpers are used to select the desired clock frequency from one of the outputs of a counter. The frequency of the signal you choose for $\overline{\text{TxC}}$ and $\overline{\text{RxC}}$ must be 1, 16, or 64 times the transmit and receive baud rate, depending on the mode in which the 8251A is initialized. Using a clock frequency higher than the



8251 Pin Functions

Pin Name	Pin Function
D7-D0	Data bus (8 bits)
$C/\overline{D}$	Control or data is to be written or read
$\overline{RD}$	Read data command
$\overline{WR}$	Write data or control command
$\overline{CS}$	Chip select
CLK	Clock pulse (TTL)
RESET	Reset
$\overline{TxC}$	Transmitter clock
TxD	Transmitter data
$\overline{RxC}$	Receiver clock
RxD	Receiver data
RxRDY	Receiver ready (has character for CPU)
TxRDY	Transmitter ready (ready for char. from CPU)
$\overline{DSR}$	Data set ready
$\overline{DTR}$	Data terminal ready
SYNDET/BD	Sync detect/break detect
$\overline{RTS}$	Request to send data
$\overline{CTS}$	Clear to send data
TxEMPTY	Transmitter empty
V_{cc}	+5-V supply
GND	Ground

(b)

FIGURE 14-3 Block diagram and pin descriptions for the Intel 8251A USART. (*a*) Block diagram. (*b*) Pin descriptions.

baud rate allows the receive shift register to be clocked at the center of the bit times rather than at leading edges. This reduces the chance of signal noise at the start of the bit time causing a read error.

The 8251A is *double-buffered*. This means that one character can be loaded into a holding buffer while another character is being shifted out of the actual transmit shift register. The TxRDY output from the 8251A will go high when the holding buffer is empty, and another character can be sent from the CPU. The TxEMPTY pin on the 8251A will go high when both the holding buffer and the transmit shift register are empty. The RxRDY pin of the 8251A will go high when a character has been shifted into the receiver buffer and is ready to be read out by the CPU. Incidentally, if a character is not read out before another character is shifted in, the first character will be overwritten and lost.

The *sync-detect/break-detect* (SYNDET/BD) pin has two uses. When the device is operating in asynchronous mode, which we are interested in here, this pin will go high if the serial data input line, RxD, stays low for more than 2 character times. This signal then indicates an intentional break in data transmission, or a break in the signal line. When programmed for synchronous data transmission, this pin will go high when the 8251A finds a specified sync character(s) in the incoming string of data bits.

The four signals connected to the box labeled MODEM CONTROL in the 8251A block diagram are handshake signals, which we described in the previous section.

INITIALIZING AN 8251A

To initialize an 8251A you must send first a mode word and then a command word to the control register address for the device. Figure 14-4 shows the formats for these words and for the 8251A status word which is read from the same address. Baud rate factor, specified by the two least significant bits of the mode word, is the ratio between the clock signal applied to the $\overline{TxC}$-$\overline{RxC}$ inputs and the desired baud rate. For example, if you want to use a $\overline{TxC}$ of 19,200 Hz and transmit data at 1200 Bd, the baud rate factor is 19,200/1200 or 16×. If bits D0 and D1 are both made 0's, the 8251A is programmed for synchronous data transfer. In this case the baud rate will be the same as the applied $\overline{TxC}$ and $\overline{RxC}$. The other three combinations for these 2 bits represent asynchronous transfer. A baud rate factor of 1 can be used for asynchronous transfer only if the transmitting system and the receiving system both use the same $\overline{TxC}$

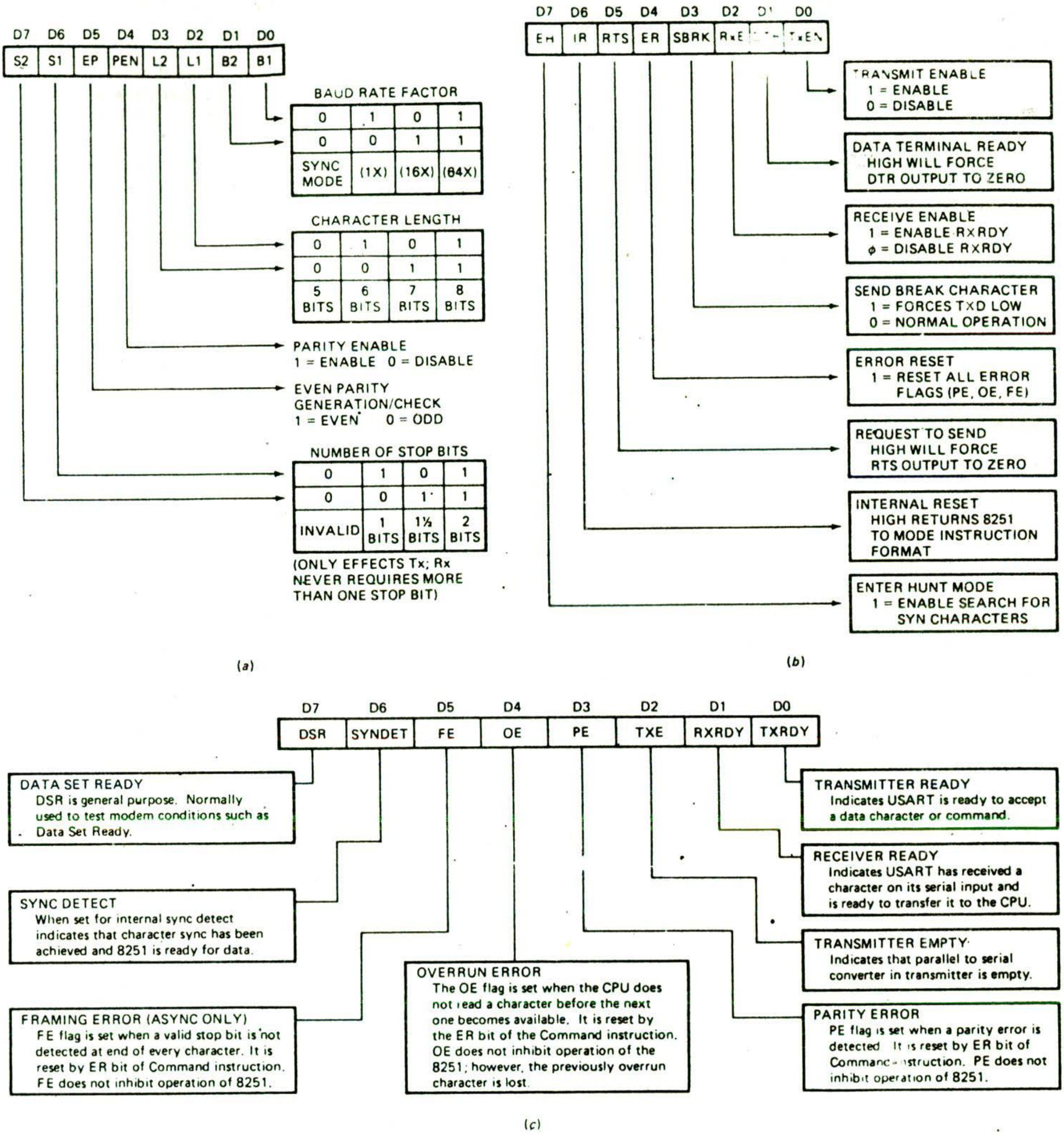

D7 D6 D5 D4 D3 D2 D1 D0
S2 S1 EP PEN L2 L1 B2 B1
BAUD RATE FACTOR
0 1 0 1
0 0 1 1
SYNC MODE (1X) (16X) (64X)
CHARACTER LENGTH
0 1 0 1
0 0 1 1
5 BITS 6 BITS 7 BITS 8 BITS
PARITY ENABLE
1 = ENABLE 0 = DISABLE
EVEN PARITY GENERATION/CHECK
1 = EVEN 0 = ODD
NUMBER OF STOP BITS
0 1 0 1
0 0 1 1
INVALID 1 BITS 1½ BITS 2 BITS
(ONLY EFFECTS Tx; Rx NEVER REQUIRES MORE THAN ONE STOP BIT)
(a)
D7 D6 D5 D4 D3 D2 D1 D0
EH IR RTS ER SBRK RxE DTR TxEN
TRANSMIT ENABLE
1 = ENABLE
0 = DISABLE
DATA TERMINAL READY
HIGH WILL FORCE DTR OUTPUT TO ZERO
RECEIVE ENABLE
1 = ENABLE RXRDY
ϕ = DISABLE RXRDY
SEND BREAK CHARACTER
1 = FORCES TXD LOW
0 = NORMAL OPERATION
ERROR RESET
1 = RESET ALL ERROR FLAGS (PE, OE, FE)
REQUEST TO SEND
HIGH WILL FORCE RTS OUTPUT TO ZERO
INTERNAL RESET
HIGH RETURNS 8251 TO MODE INSTRUCTION FORMAT
ENTER HUNT MODE
1 = ENABLE SEARCH FOR SYN CHARACTERS
(b)
D7 D6 D5 D4 D3 D2 D1 D0
DSR SYNDET FE OE PE TXE RXRDY TXRDY
DATA SET READY
DSR is general purpose. Normally used to test modem conditions such as Data Set Ready.
SYNC DETECT
When set for internal sync detect indicates that character sync has been achieved and 8251 is ready for data.
FRAMING ERROR (ASYNC ONLY)
FE flag is set when a valid stop bit is not detected at end of every character. It is reset by ER bit of Command instruction. FE does not inhibit operation of 8251.
OVERRUN ERROR
The OE flag is set when the CPU does not read a character before the next one becomes available. It is reset by the ER bit of the Command instruction. OE does not inhibit operation of the 8251; however, the previously overrun character is lost.
TRANSMITTER READY
Indicates USART is ready to accept a data character or command.
RECEIVER READY
Indicates USART has received a character on its serial input and is ready to transfer it to the CPU.
TRANSMITTER EMPTY
Indicates that parallel to serial converter in transmitter is empty.
PARITY ERROR
PE flag is set when a parity error is detected. It is reset by ER bit of Command instruction. PE does not inhibit operation of 8251.
(c)

FIGURE 14-4 Formats of 8251A mode, command, and status words. (*a*) Mode word. (*b*) Command word. (*c*) Status word. (*Intel Corporation*)

and $\overline{\text{RxC}}$. The character length specified by bits D2 and D3 in the mode word includes only the actual data bits, not the start bit, parity bit, or stop bit(s). If parity is disabled, no parity bit is inserted in the transmitted bit string. If the 8251A is programmed for 5, 6, or 7 data bits, the extra bits in the data character byte read from the device will be 0's.

After you send a mode word to an 8251A, you must then send it a command word. A 1 in the least significant bit of the command word enables the transmitter section of the 8251A and the TxRDY output. When enabled, the 8251A TxRDY output will be asserted high if the $\overline{\text{CTS}}$ input has been asserted low, and the transmitter holding buffer is ready for another character from the CPU. The TxRDY signal can be connected to an interrupt input on the CPU or an 8259A, so that characters to be transmitted can be sent to the 8251A on an interrupt basis. When a character is written to the 8251A data address, the TxRDY signal will go low and remain low until the holding buffer is again ready for another character. Putting a 1 in bit D1 of the command word will cause the DTR output of the 8251A to be asserted low. As we explained before, this signal is used to tell a modem that a terminal or computer is operational. A 1 in bit D2 of the command word enables the RxRDY output pin of the 8251A. If enabled, the RxRDY pin will go high when the 8251A has a character in its receiver buffer ready to be read. This signal can be connected to an interrupt input so that characters can be read in on an interrupt basis. The RxRDY output is reset when a character is read from the 8251A.

Putting a 1 in bit D3 of the command word causes the 8251A to output a character of all 0's, which is called a break character. A break character is sometimes used to indicate the end of a block of transmitted data. Sending a command word with a 1 in bit D4 causes the 8251A to reset the parity, overrun, and framing error flags in the 8251A status register. The meanings of these flags are explained in Figure 14-4c. A 1 in bit D5 of the command word will cause the 8251A to assert its request-to-send (RTS) output low. This signal, remember, is sent to a modem to ask whether the modem and the receiving system are ready for a data character to be sent.

Putting a 1 in bit D6 of the command word causes the 8251A to be internally reset when the command word is sent. After a software reset command is sent in this way, a new mode word must be sent. Later we will show you how this is used.

The D7 bit in the command word is only used when the device is operating in synchronous mode. A command word with a 1 in this bit position tells the 8251A to look for specified sync character(s) in a stream of bits being shifted in. If the 8251A finds the specified sync character(s), it will assert its SYNDET/BD pin high. We will discuss this more in the synchronous data communication section of this chapter.

Figure 14-5 shows an example of the instruction sequence you can use to initialize an 8251A. This sequence is somewhat lengthy for two reasons. First, the 8251A does not always respond correctly to a hardware reset on power-up. Therefore, a series of software commands must be sent to the device to make sure it is reset properly before the desired mode and command words are sent. The device is put into a known state by writing 3 bytes of all 0's to the 8251A control register address, and then it is reset by sending a control word with a 1 in bit D6. After this reset sequence the desired mode and control words can be sent to 8251A. The 8251A distinguishes a command word from a mode word by the order in which they are sent to the device. After reset, a mode word must be sent to the command address. Any words sent to the command address after the mode word will be treated as command words until the device is reset.

```
; 8086 instructions to initialize the 8251A on an
; SDK-86 board

   MOV  DX, 0FFF2H ; point at command register address
   MOV  AL, 00H    ; send 0's to guarantee device is
   OUT  DX, AL     ; in the command instruction format
   MOV  CX, 2      ; before the RESET command is
D0:LOOP D0         ; issued and delay after sending
   OUT  DX, AL     ; each command instruction
   MOV  CX, 2
D1:LOOP D1
   OUT  DX, AL
   MOV  CX, 2
D2:LOOP D2
   MOV  AL, 40H     ; Sent internal reset command to
   OUT  DX, AL      ; return device to idle state
   MOV  CX, 2       ; Load delay constant
D3:LOOP D3          ; and delay
   MOV  AL,11001110B; Load mode control word & send it
   OUT  DX, AL
; 1 1 0 0 1 1 1 0    Mode Word
;  \ \ \ \ \ \ \_\___baud rate factor of 16x
;   \ \ \ \ \_\______character length of 8 bits
;     \ \ \_\ _______parity disabled
;      \_\___________2 stop bits

   MOV  CX, 2        ; and delay
D4:LOOP D4
   MOV  AL,00110111B ; Load command word and send it
   OUT  DX, AL
; 0 0 1 1 0 1 1 1     Command word
; \ \ \ \ \ \ \ \_____Transmit enable
;  \ \ \ \ \ \ \______Data terminal ready, DTR will
;   \ \ \ \ \ \       output 0
;    \ \ \ \ \ \______Receive enable
;     \ \ \ \ \_______Normal operation
;      \ \ \ \________Reset all error flags
;       \ \ \_________RST output 0, request to send
;        \ \__________Do not return to mode
;         \           instruction form
;          \__________Disable hunt mode
```

FIGURE 14-5 Instruction sequence for 8251A initialization.

The second factor which lengthens this initialization is the *write-recovery* time T_{RV} of the 8251A. According to the data sheet, the 8251A requires a worst-case recovery time of 16 cycles of the clock signal connected to the CLK input. On the SDK-86 board the PCLK signal, which is the same as the processor clock frequency, is connected to the CLK input of the 8251A. Therefore, for the SDK-86 board, the required write-recovery time corresponds to 16 processor clock cycles. What all this means is that you have to delay this many clock cycles between successive initialization byte writes to the 8251A. A simple way to produce the required delay and a margin of safety is to load CX with 0002 and count it down with the LOOP instruction. The MOV CX,0002 instruction takes 4 clock cycles, the first execution of the LOOP instruction takes 17 clock cycles, and the last execution of the LOOP instruction takes 5 cycles. The 8 cycles required for the OUT instruction, which writes the control words, also count as part of the time between writes, so the sum of all these is more than enough. When writing data characters to an 8251A, you don't have to worry about this recovery time, because a new character will not be written to the 8251A until the previous character has been shifted out. This shifting, of course, requires much more time than T_{RV}.

The comments in Figure 14-5 explain the meanings of the bits in the mode and control words used in this example. Once the 8251A is initialized as shown, new control words can be sent at any time to, for example, reset the error flags. Now let's look at how characters are sent to and read from an 8251A.

SENDING AND RECEIVING CHARACTERS WITH AN 8251A

Data characters can be sent to and read from the 8251A on an interrupt basis or on a polled basis. To send characters on an interrupt basis, the TxRDY pin of the 8251A is connected to an interrupt input on the processor of an 8259A priority-interrupt controller. The transmitter and the TxRDY output are enabled by putting a 1 in bit D1 of the control word sent to the 8251A during initialization. When the $\overline{\text{CTS}}$ input of the 8251A is asserted low and the 8251A buffer is ready for a character, the TxRDY pin will go high. If the processor and 8259A interrupt path is enabled, the processor will go to an interrupt-service procedure, which writes a data character to the 8251A data address. Writing the data character causes the 8251A to reset its TxRDY output until the buffer is again ready to receive a character. A counter can be used to keep track of how many characters have been sent.

In a similar manner characters can be read from an 8251A on an interrupt basis. In this case the RxRDY output of the 8251A is connected to an interrupt input of the processor or an 8259A, and this output is enabled by putting a 1 in bit D2 of the command word sent during initialization. When a character has been shifted into the 8251A and the character is in the receiver buffer ready to be read, the RxRDY pin will go high. If the interrupt chain through the 8259A and the processor is enabled, the processor will go to an interrupt procedure which reads in the data character. Reading a data character from the 8251A causes it to reset the RxRDY output signal. This signal will stay low until another character is ready to be read.

To send characters to an 8251A on a polled basis, the 8251A status register is read and checked over and over until the TxRDY bit (D0) is found to be a 1. In most systems you also want to check bit D7 of the status register to make sure the $\overline{\text{DSR}}$ input of the 8251A has been asserted by a signal from, for example, a modem. When the required bit(s) of the status register are all high, a data character is then written to the 8251A data address. Figure 14-6*a* shows the instruction sequence needed to do this. Note that the status register has the same internal address as the control register. Also note that both an AND and a CMP operation must be done to determine when the two desired bits are both high. Writing a data character to the 8251A resets the TxRDY bit in the status register.

```
; Instructions for transmitting data using an
; SDK-86 8251A using polling method

  MOV DX, OFFF2H          ; Point at control register
TEST1:                    ; address
  IN  AL, DX              ; Read status
  AND AL, 10000001B       ; and check status of
;           \______\___data set ready & transmit ready
  CMP AL, 10000001B       ; Is it ready?
  JNE TEST1               ; Continue to poll if not ready
  MOV DX, OFFF0H          ; otherwise point at data address
  MOV AL, DATA_TO_SEND    ; Load data to send
  OUT DX, AL              ; and send it
```

(a)

```
; Instructions for receiving data with an
; SDK-86 8251A using polling method

  MOV DX, OFFF2H          ; Point at control register
TEST2:                    ; address
  IN  AL, DX              ; Read status
  AND AL, 00000010B       ; and check status of RxRdy
  JZ  TEST2               ; Continue to poll if not ready
  MOV DX, OFFF0H          ; otherwise point at data
  IN  AL, DX              ; address and get data
```

(b)

FIGURE 14-6 Instruction sequences for transmitting and receiving with an 8251A on a polled basis. (*a*) Transmit. (*b*) Polled.

Reading a character from the 8251A on a polled basis is a similar process, except that the RxRDY bit (D1) of the status register is polled to determine when a character is ready to be read. When bit D1 is found high, a character is read in from the 8251A data address. Figure 14-6*b* shows the instruction sequence for this. Status register bits D3, D4, and D5 can be checked to see if a parity error, overrun error, or framing error has occurred. If an error has occurred, a message to retransmit the data can be sent to the transmitting system.

The next step in our journey into serial-data communications is to discuss the signal standards used to connect the serial inputs and outputs of UARTS to modems and other serial devices.

SERIAL-DATA TRANSMISSION METHODS AND STANDARDS

In the last section we showed you how a UART or USART is used to interface microcomputer buses with serial-data communication lines. The TTL signals output by a USART, however, are not suitable for transmission over long distances, so these signals are converted to some other form to be transmitted. In this section of the chapter we discuss devices and signal types commonly used to send serial-data signals over long distances.

Aside from drum beats in the jungle, one of the earliest forms of serial-data communication was the telegraph. In a telegraph, pressing a key at one end of a signal line causes a current to flow through the line. When this current reaches the receiving end of the line, it activates a solenoid (sounder), which produces a sound. Letters and numbers are sent using the familiar Morse code or some other convenient code. After a hundred years or so, the telegraph key and sounder evolved into the teletypewriter. A teletypewriter terminal has a typewriter-style keyboard so that the user can simply press a key to send a desired letter or number code. A teletype terminal also has a print mechanism which prints out characters as they are received. Most teletypes use a current to represent a 1 and no current to represent a 0. We start this section by briefly describing the old current-loop standards; then we go on to newer methods.

20- AND 60-mA CURRENT LOOPS

In teletypewriters or other current-signal systems, some manufacturers use a nominal current of 20 mA to represent a 1, or mark, and no current to represent a space, or 0. Other manufacturers use a nominal current of 60 mA to represent a 1 and no current to represent a 0. The actual current in a specific system may be considerably different from the nominal value.

Sheet 9 of Figure 7-8 shows circuitry which can be used to interface current type signals with the TTL input and output of an 8251A USART on the SDK-86 board. With the jumpers in place as shown, a high on the TxD output of the 8251A will produce a low on the PNP transistor. This will turn the transistor on and cause a positive current to flow out the TTY TX line. Inside a teletypewriter this current flows through an electromagnet and back to the TTY TX RET. To send a data bit, the teletypewriter opens or closes a switch in a current path. The current for this path in the SDK-86 circuitry is supplied from +5 V through R10 to the TTY RX RET line. Think of the key mechanism of the teletypewriter as a simple switch connected between pins 24 and 12 of J7 on the circuit. When the switch is closed the current flows back on the TTY RX line and through R3 to −12 V. The current flowing through R3 will produce a legal TTL high logic level on the input of the 74LS14 inverter. This high signal passes through two inverters and produces a high on the RxD input of the 8251A.

RS-232C Serial Data Standard

OVERVIEW

In the 1960s as the use of timeshare computer terminals became more widespread, modems were developed so that terminals could use phone lines to communicate with distant computers. As we stated earlier, modems and other devices used to send serial data are often referred to as *data communication equipment* or DCE. The terminals or computers that are sending or receiving the data are referred to as *data terminal equipment* or DTE. In response to the need for signal and handshake standards between DTE and DCE, the Electronic Industries Association (EIA) developed EIA standard *RS-232C*. This standard describes the function of 25 signal and handshake pins for serial-data transfer. It also describes the voltage levels, impedance levels, rise and fall times, maximum bit rate, and maximum capacitance for these signal lines. Before we work our way through the 25 pin functions, we will take a brief look at some of the other hardware aspects of RS-232C.

RS-232C specifies 25 signal pins, and it specifies that the DTE connector should be a male and the DCE connector should be a female. A specific connector is not given, but the most commonly used connectors are the DB-25P male shown in Figure 14-7*a*. For systems where many of the 25 pins are not needed, a 9-pin DIN connector such as the DE-9P male connector shown in Figure 14-7*b* is used. When you are wiring up these connectors, it is important to note the order in which the pins are numbered.

The voltage levels for all RS-232C signals are as follows. A logic high, or mark, is a voltage between −3 V and −15 V under load (−25 V no load). A logic low or space is a voltage between +3 V and +15 V under load (+25 V no load). Voltages such as ±12 V are commonly used.

RS-232C TO TTL INTERFACING

Obviously a USART such as the 8251A is not directly compatible with RS-232C signal levels. Sheet 9 of the SDK-86 schematics in Figure 7-8 shows one way to interface TTL signals of the 8251A to RS-232C signal levels. If the jumpers shown are removed and the jumpers shown in the jumper table under CRT are inserted, the circuit will produce and accept RS-232C signals.

> NOTE: This is the jumpering needed to prepare the SDK-86 board for downloading programs from an IBM PC or other computer. Here's how it works.

With a jumper between the points numbered 7 and 8, a high on the TxD output of the 8251A produces a high on the base of the transistor, which turns it off. With points numbered 9 and 10 jumpered, the CR TX line will then be pulled to −12 V, which is a legal high or marking condition for RS-232C. A low on the TxD output of the 8251A will turn on the transistor and pull the CR TX line to +5 V, which is a legal low or space condition for RS-232C.

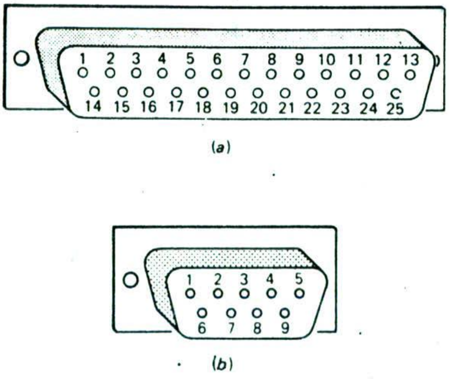


FIGURE 14-7 Connectors often used for RS-232C connections. (*a*) DB-25P 25-pin male. (*b*) DE-9P 9-pin male DIN connector.

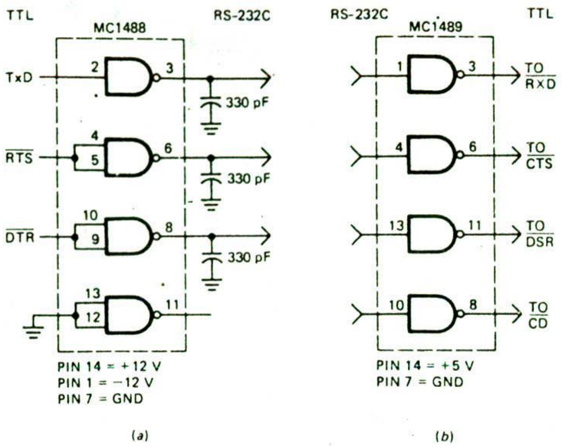


FIGURE 14-8 TTL to RS-232C to TTL signal conversion. (*a*) MC1488 used to convert TTL to RS-232C. (*b*) MC1489 used to convert RS-232C to TTL.

Another, more standard way to interface between RS-232C and TTL levels is with MC1488 quad TTL-to-RS-232C drivers and MC1489 quad RS-232C-to-TTL receivers shown in Figure 14-8. The MC1488s require + and − supplies, but the MC1489s require only +5 V. Note the capacitor to ground on the outputs of the MC1488 drivers. To reduce cross talk between adjacent wires, the rise and fall times for RS-232C signals are limited to 30 V/μs. Also note that the RS-232C handshake signals such as $\overline{\text{RTS}}$ are active low. Therefore, if one of these signals is asserted, you will find a positive voltage on the actual RS-232C signal line when you check it during troubleshooting. Now let's look at the RS-232C pin descriptions.

RS-232C SIGNAL DEFINITIONS

Figure 14-9 shows the signal names, signal direction, and a brief description for each of the 25 pins defined for RS-232C. For most applications only a few of these pins are used, so don't get overwhelmed. Here are a few additional notes about these signals.

First note that the signal direction is specified with respect to the DCE. This convention is part of the standard. We have found it very helpful to put arrowheads on all signal lines, as shown in Figure 14-2, when we are drawing circuits for connecting RS-232C equipment.

Next observe that there is both a chassis ground (pin 1) and a signal ground (pin 7). To prevent large ac-induced ground currents in the signal ground, these two should be connected together only at the power supply in the terminal or the computer.

The TxD, RxD, and handshake signals shown with common names in Figure 14-9 are the ones most often used for simple systems. We gave an overview of their use in the introduction to this section of the chapter and will discuss them further in a larger section of the chapter on modems. These signals control what is called the *primary* or *forward* communications channel of the modem. Some modems allow communication over a *secondary* or *backward* channel, which operates in the reverse direction from the forward channel and at a much lower baud rate. Pins 12, 13, 14, 16, and 19 are the data and handshake lines for this backward channel.

Pins 15, 17, 21, and 24 are used for synchronous data

PIN NUMBERS FOR 9 PINS	PIN NUMBERS FOR 25 PINS	COMMON NAME	RS-232C NAME	DESCRIPTION	SIGNAL DIRECTION ON DCE
	1		AA	PROTECTIVE GROUND	–
3	2	TXD	BA	TRANSMITTED DATA	IN
2	3	RXD	BB	RECEIVED DATA	OUT
7	4	$\overline{\text{RTS}}$	CA	REQUEST TO SEND	IN
8	5	$\overline{\text{CTS}}$	CB	CLEAR TO SEND	OUT
6	6	$\overline{\text{DSR}}$	CC	DATA SET READY	OUT
5	7	GND	AB	SIGNAL GROUND (COMMON RETURN)	–
1	8	$\overline{\text{CD}}$	CF	RECEIVED LINE SIGNAL DETECTOR	OUT
	9		–	(RESERVED FOR DATA SET TESTING)	–
	10		–	(RESERVED FOR DATA SET TESTING)	–
	11			UNASSIGNED	–
	12		SCF	SECONDARY RECEIVED LINE SIGNAL DETECTOR	OUT
	13		SCB	SECONDARY CLEAR TO SEND	OUT
	14		SBA	SECONDARY TRANSMITTED DATA	IN
	15		DB	TRANSMISSION SIGNAL ELEMENT TIMING (DCE SOURCE)	OUT
	16		SBB	SECONDARY RECEIVED DATA	OUT
	17		DD	RECEIVER SIGNAL ELEMENT TIMING (DCE SOURCE)	OUT
	18			UNASSIGNED	–
	19		SCA	SECONDARY REQUEST TO SEND	IN
4	20	$\overline{\text{DTR}}$	CD	DATA TERMINAL READY	IN
	21		CG	SIGNAL QUALITY DETECTOR	OUT
9	22		CE	RING INDICATOR	OUT
	23		CH/CI	DATA SIGNAL RATE SELECTOR (DTE/DCE SOURCE)	IN/OUT
	24		DA	TRANSMIT SIGNAL ELEMENT TIMING (DTE SOURCE)	IN
	25			UNASSIGNED	–

FIGURE 14-9 RS-232C pin names and signal directions.

communication. We will tell you a little more about these in the section of the chapter on modems. Next we want to show you some of the tricks in connecting RS-232C-"compatible" equipment.

CONNECTING RS-232C-COMPATIBLE EQUIPMENT

A major point we need to make right now is that you can seldom just connect together two pieces of equipment, described by their manufacturers as RS-232C compatible, and expect them to work the first time. There are several reasons for this. To give you an idea of one of the reasons, suppose that you want to connect the terminal in Figure 14-2 directly to the computer rather than through the modem-modem link. The terminal and the computer probably both have DB-25-type connectors so that, other than a possible male-female mismatch, you might think you could just plug the terminal cable directly into the computer. To see why this doesn't work, hold your fingers over the modems in Figure 14-2 and refer to the pin numbers for the RS-232C signals in Figure 14-9. As you should see, both the terminal and the computer are trying to output data (TxD) from their number 2 pins to the same line. Likewise, they are both trying to input data (RxD) from the same line on their number 3 pins. The same problem exists with the handshake signals. RS-232C drivers are designed so that connecting the lines together in this way will not destroy anything, but connecting outputs together is not a productive relationship. A solution to this problem is to make an adapter with two connectors so that the signals cross over, as shown in Figure 14-10*a*. This crossover connection is often called a *null modem*. We have again put arrowheads on the signals in Figure 14-10*a* to help you keep track of the direction for each. As you can see in the figure, the TxD from the terminal now sends data to the RxD input of the computer. Likewise, the TxD from the computer now sends data to the RxD input of the terminal as desired. The handshake signals also are crossed over so that each handshake output signal is connected to the corresponding input signal.

A second reason that you can't just plug RS-232C-compatible equipment together and expect it to work is that a partial implementation of RS-232C is often used to communicate with printers, plotters, and other computer peripherals besides modems. These other peripherals may be configured as DCE or as DTE. Also, they may use all, some, or none of the handshake signals. As an example of this, suppose that you want to connect the RS-232C port on the IBM PC asynchronous communication board to the serial port on the SDK-86 so that you can download object-code programs.

The IBM PC asynchronous board is configured as DTE, so TxD is on pin 2, RxD is on pin 3, $\overline{\text{RTS}}$ is on pin 4, $\overline{\text{CTS}}$ is on pin 5, $\overline{\text{DTR}}$ is on pin 20, $\overline{\text{DSR}}$ is on pin 6, and carrier detect ($\overline{\text{CD}}$) is on pin 8. In order for the IBM board to be able to transmit and receive, its $\overline{\text{CTS}}$, $\overline{\text{DSR}}$, and $\overline{\text{CD}}$ inputs must be asserted. The BIOS software asserts the $\overline{\text{DTR}}$ and $\overline{\text{RTS}}$ outputs.

Now take another look at sheet 9 of the SDK-86 schematics in Figure 7-6 to see how the data and handshake signals are connected there. For communicating with RS-232C-type equipment, the SDK-86 board is jumpered as shown in the jumper table column labeled "stand-alone CRT." The output data on CRT TX then connects to pin 3 of connector J7, a DB-25S-type connector. This corresponds to the RxD on the IBM connector, so no crossover is needed. Likewise, the CRT RX of the SDK corresponds to the TxD of the IBM board, so this is also a straight-through connection. The handshake signals here are another story.

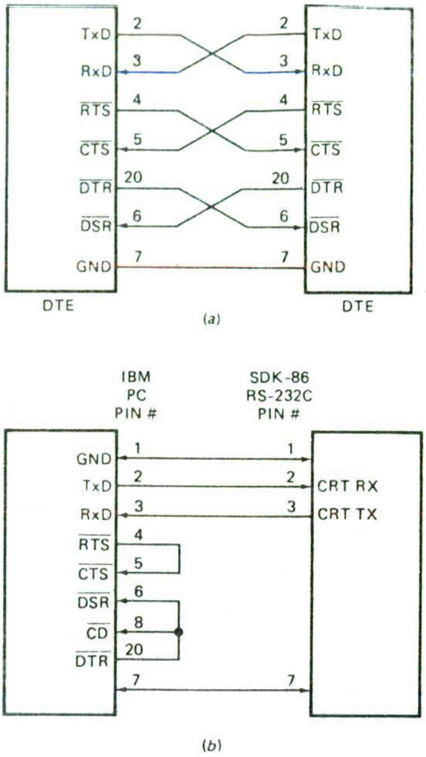


FIGURE 14-10 Nonmodem RS-232C connections. (*a*) Null modem for connecting two RS-232C data terminal–type devices. (*b*) IBM PC or PS/2 serial port to SDK-86 serial port connection.

The $\overline{\text{RTS}}$ of the SDK-86 is simply looped into the $\overline{\text{CTS}}$, so $\overline{\text{CTS}}$ will automatically be asserted when $\overline{\text{RTS}}$ is asserted by the 8251A. Therefore, neither of these signals is available for external handshaking. The $\overline{\text{DTR}}$ output of the 8251A on the SDK board is used for a teletypewriter function and does not connect to the normal RS-232C $\overline{\text{DTR}}$ pin number, so it is not available either. The $\overline{\text{DSR}}$ input of the 8251A is connected to the RxD input so that it will be asserted when a start bit comes in on the serial-data line, but this line is also not available for handshaking with external devices. Therefore, the problem here is that the SDK-86 is not set up to supply the handshake signals needed by the IBM PC serial board. Figure 14-10*b* shows the connections you make to solve this problem so the PC can talk to the SDK-86. The PC $\overline{\text{RTS}}$ line on pin 4 is jumpered on the connector to its $\overline{\text{CTS}}$ line on pin 5, so that $\overline{CTS}$ will automatically be asserted when $\overline{RTS}$ is asserted. Pins 6, 8, and 20 are also jumpered together on the connector so that when the PC asserts its $\overline{DTR}$ output on pin 20, the $\overline{DSR}$ input and the $\overline{CD}$ input will automatically be asserted. These connections do not provide for any hardware handshaking. They are necessary just to get the PC and the SDK-86 to talk to each other.

The point here is that whenever you have to connect RS-232C-compatible devices such as terminals, serial printers, etc., get the schematic for each and work your way through the connections one pin at a time. Make sure that an output on one device goes to the appropriate input on the other device. Sometimes you have to look at the actual drivers and receivers on the schematic to determine which pins on the connector are outputs and which are inputs. This is necessary because some manufacturers label an output pin connected to pin 3 as RxD, indicating that this signal goes to the RxD input of the receiving system.

If you do not have schematics for the RS-232C equipment you are trying to connect, you can often use a *breakout box* to determine the correct connections. You insert the breakout box in series with the connecting cable and LEDs on the box indicate which lines are outputs and which lines are inputs. By throwing switches on the box, you can try different connection combinations until data transfers correctly.

RS-423A and RS-422A

RS-423

A major problem with RS-232C is that it can only transmit data reliably for about 50 ft (16.4 m) at its maximum rate of 20,000 Bd. If longer lines are used, the transmission rate has to be drastically reduced. This limitation is caused by the open signal lines with a single common ground that are used to RS-232C.

Another EIA standard which is an improvement over RS-232C is *RS-423A*. This standard specifies a low-impedance single-ended signal which can be sent over 50-Ω coaxial cable and partially terminated at the receiving end to prevent reflections. Figure 14-11 shows how an MC3487 driver and MC3486 receiver can be connected to produce the required signals. A logic high in this standard is represented by the signal line being between 4 and 6 V negative with respect to ground, and a logic low is represented by the signal line being 4 to 6 V positive with respect to ground.

The RS-423 standard allows a maximum data rate of 100,000 Bd over a 40-foot line or a maximum baud rate of 1000 Bd on a 4000-foot line.

RS-422A

A still-newer standard for serial data transfer, *RS-422A* specifies that each signal will be sent differentially over two adjacent wires in a ribbon cable or a twisted pair of wires, as shown in Figure 14-12*a*, page 498.

The term differential in this standard means that the signal voltage is developed between the two signal lines rather than between a signal line and ground as in RS-232C and RS-423. In RS-422A a logic high is transmitted by making the "b" line more positive than the "a" line. A logic low is transmitted by making the a line more positive than the b line. The voltage difference between the two lines must be greater than 0.4 V but less than 12 V. Typical drivers such as the MC3487 shown in Figure 14-12a produce a differential voltage of about 2 V. The center or common-mode voltage on the lines must be between −7 V and +7 V. RS-422A specifies signal rise and fall times of 20 ns or 0.1 multiplied by the time for 1 bit, whichever is greater.

Figure 14-12*b* shows the relationship between maximum cable length and baud rate for RS-422A line. As we hope you can see in this graph, the maximum data rate for RS-422A lines ranges from 10 million Bd on a line 40 ft long to 100,000 Bd on a 4000-foot line. The reason that the data rates are so much higher than for RS-423 lines is that the differential line functions as a fully terminated transmission line. Common 24-gage twisted-pair wire has a Z_0 of about 100 Ω, so the line can be terminated with a matching 100-Ω resistor connected between the signal lines. A more common termination method, however, is to use a 50-Ω resistor from each signal line to ground as shown in Figure

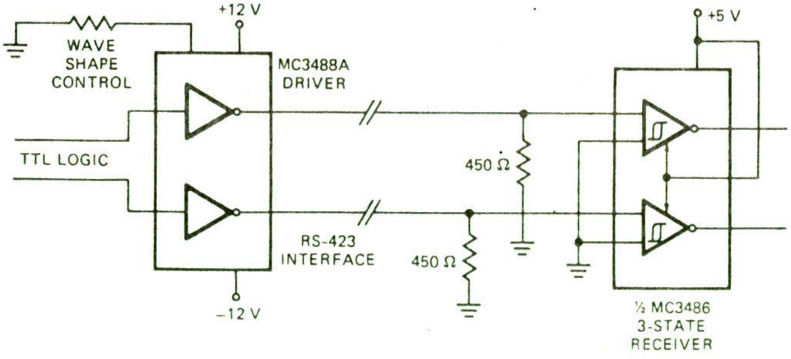


FIGURE 14-11 MC3488A driver and MC3486 receiver used for RS-423 signal transmission.

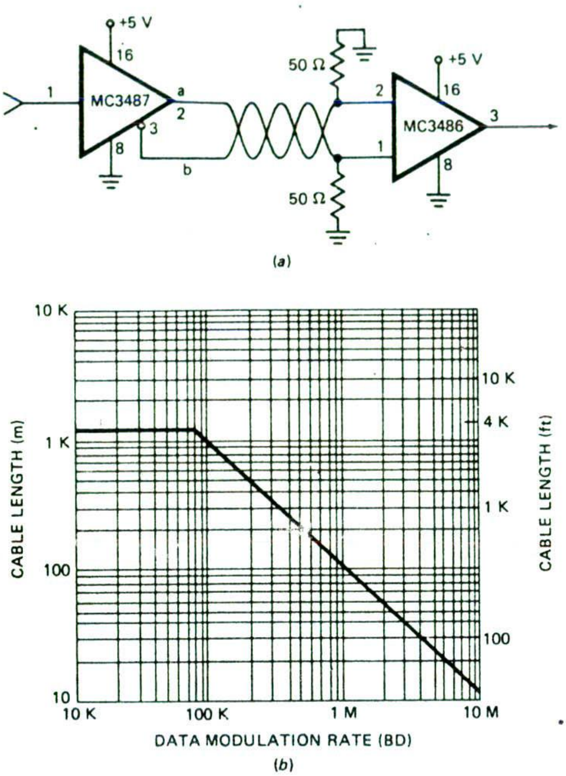


FIGURE 14-12 (a) MC3487 driver and MC3486 receiver used for RS-422A differential signal. (*b*) Maximum line length versus baud rate for RS-422A signal lines.

14-12*a*. This method helps keep the two signal lines balanced.

A further advantage of differential signal transmission is that any electrical noise induced in one signal line will be induced equally in the other signal line. A differential line receiver such as the MC3486 shown in Figure 14-12*a* responds only to the voltage difference between its two inputs, so any noise voltage that is induced equally on the two inputs will not have any effect on the output of the differential receiver.

The RS-422A and RS-423A standards do not specify connector pin numbers or handshake signals the way the RS-232C does. An additional EIA standard called *RS-449* does this for the two. RS-449 specifies 37 signal pins on a main connector and 9 additional pins on an optional connector. The signals on these connectors are a superset of the RS-232C signals so adapters can be used to interface RS-232C equipment with RS-449 equipment.

Now that we have discussed the signals commonly used to interface a computer to a modem, let's take a closer look at how modems transmit signals over standard phone lines.

Modems

INTRODUCTION

As we described in a previous section, a modulator-demodulator, or modem, sends digital 1's and 0's over standard phone lines as modulated tones. The frequency of the tones is within the bandpass of the lines. Two organizations are responsible for most of the current standards for modem modulation methods and transmissions rates. Older modems in the United States were based on de facto standards from Bell Telephone Company. Examples of these standards are the Bell types 103, 202, 308, and 212A. In the United States modem standards are now handled by the Telecommunications Industry Association, which works very closely with the *Comité Consultatif Internationale Téléphonique et Télégraphique* (CCITT), which is part of the International Telecommunications Union. CCITT standards which relate to modems start with a V. Examples are the V.22 bis, which is a 2400-bit/s modem standard, and the V.29, which is a 9600-bit/s modem standard. As we discuss modem modulation techniques in the following section, we will describe these and other standards in greater detail.

INTRODUCTION TO MODEM MODULATION

To represent digital 1's and 0's a modulator changes some characteristic of an audio signal which has a frequency within the bandwidth of the phone lines. An important point to keep in mind as you read through the following section is that the maximum rate at which the audio tone can be modulated is one-half the bandwidth of the transmission line. If, for example, we assume that the worst-case bandwidth of a two-wire phone line is 2400 Hz, then the maximum modulation rate for a half-duplex signal on the line is 1200 Bd. For full-duplex communication, half the bandwidth is used for transmission in each direction, so the maximum modulation rate for each direction on a two-wire phone line is 600 baud. In a 4-wire phone line which has separate wires for each direction, the maximum modulation rate for each direction is 1200 Bd. One of the goals of this section is to show you the modulation techniques that are used to overcome these basic limitations.

The major forms of modulation used are *amplitude*, *frequency-shift keying* (FSK), *phase-shift keying* (PSK), and *multiple carrier*.

As the name implies, amplitude modulation changes the amplitude of the transmitted tone. One common way of doing this is to turn a 387-Hz tone on to represent a 1 and turn the tone off to represent a turn the tone off to represent a 0, as shown in Figure 14-13. In other systems that we discuss later, the tone is always present, but its amplitude is changed between two or more values. Amplitude modulation is used only for very low speed reverse-channel transmission or in conjunction with some other type modulation such as phase modulation.

FREQUENCY-SHIFT KEYING MODULATION

Frequency-shift keying or FSK modulation uses one tone to represent a 0 and another tone to represent a 1, as

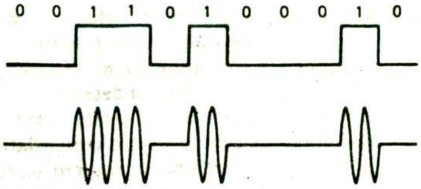


FIGURE 14-13 Representation of digital 1's and 0's with amplitude-modulated sine waves.

shown in Figure 14-14. In order to allow full-duplex communication, four different frequencies are often used. An old standard, the Bell 103A, 300-Bd FSK modem, for example, uses 2025 Hz for a 0 and 2225 Hz for a 1 in one direction, and 1070 Hz for a 0 and 1270 Hz for a 1 in the other direction. Another standard, the Bell 202 modem, permits half-duplex communication at 1200 baud. The 202 uses 1200 Hz to represent a 0 and 1700 Hz to represent a 1 for the main channel. Different versions of the 202 may also have either a 5-bit/s amplitude-modulated back channel or a 150-bit/s FSK back channel which uses 387 Hz for a 0 and 487 Hz for a 1.

As we discussed before, simple modulation such as FSK is limited to half-duplex operation at 1200 Bd on two-wire phone lines or 1200 Bd full-duplex on four-wire phone lines. For higher bit rates some type of phase-shift modulation is used.

PHASE-SHIFT MODULATION VARIATIONS

In the simplest form of phase-shift modulation called *differential phase-shift modulation* or DPSK, the phase of a constant-frequency sine-wave carrier of perhaps 1700 Hz is shifted by 180° to represent a change in the data from a 1 to a 0 or a change in the data from a 0 to a 1. Figure 14-15*a* shows an example of this. As the digital data changes from a 0 to a 1, near the left edge of the figure, the phase of the signal is shifted by 180°. When data changes from a 1 to a 0, the phase of the carrier is again shifted by 180°. For the next section of the digital data where the data stays 0 for 3 bit times, the phase of the carrier is not changed. Likewise, in a later section of the waveform where the data remains at a one level for 2 bit times, the phase of the carrier is not changed. The phase of the carrier then is shifted by 180° only when the data line changes from a 1 to a 0 or from a 0 to a 1.

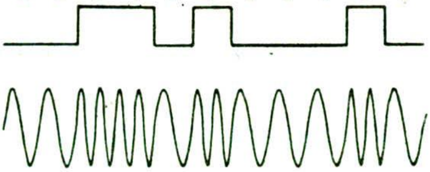
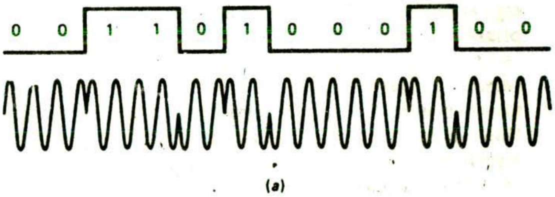


FIGURE 14-14 Representation of digital 1's and 0's with two different frequencies (FSK).

The simple phase-shift modulation shown in Figure 14-15*a* has no real advantage over FSK as far as maximum bit rate is concerned. However, by using additional phase angles besides 180°, 2 or more data bits can be sent with one phase change. Figure 14-15*b* shows how the value of 2 bits can be represented by four different phase shifts. If, for example, the value of a ***dibit***, or 2 bits taken together, is 00, the phase of the carrier will be shifted 90° to represent that dibit. The trick here is that the phase of the carrier only has to shift once for each group of 2 transmitted bits.

Remember from a previous discussion that the baud rate limitation we are trying to overcome is the rate at which the carrier is changing. In this case the number of data bits per second is twice the baud rate. Bell 212A- and CCITT V.22-type modems use this scheme to transmit 1200 bits/s at an effective baud rate of only 600 Bd. Two carrier frequencies, 1200 Hz and 2400 Hz, are used to permit full-duplex operation at this rate.

A more complex phase-shift modulation scheme called *quaternary amplitude modulation* or QAM enables V.22-bis-type modems to transmit full-duplex data at 2400 bit/s over two-wire phone lines. V.29-type modems also use this type modulation to transmit half-duplex 9600-bit/s data from facsimile (FAX) machines. QAM uses 12 different phase angles and three different amplitudes to encode 4 data bits in each modulation change. Each group of 4 data bits is referred to as a ***quadbit***. A phase-amplitude graph such as that shown in Figure 14-16, page 500, is often used to represent the phase and amplitude values for each of the 16 possible quadbits. Incidentally, the pattern of phase-amplitude points in



GRAY CODE DIBIT VALUE	DEGREES OF PHASE SHIFT
0 0	0
0 1	90
1 1	180
1 0	270

(b)

GRAY CODE TRIBIT VALUE	DEGREES OF PHASE SHIFT
0 0 1	22.5
0 0 0	67.5
0 1 0	112.5
0 1 1	157.5
1 1 1	202.5
1 1 0	247.5
1 0 0	292.5
1 0 1	337.5

(c)

FIGURE 14-15 Phase-shift modulation. (a) Waveforms for simple phase-shift modulation. (*b*) Set of phase shifts used to represent four possible dibit combinations. (c) Set of phase shifts used to represent eight possible tribit combinations.

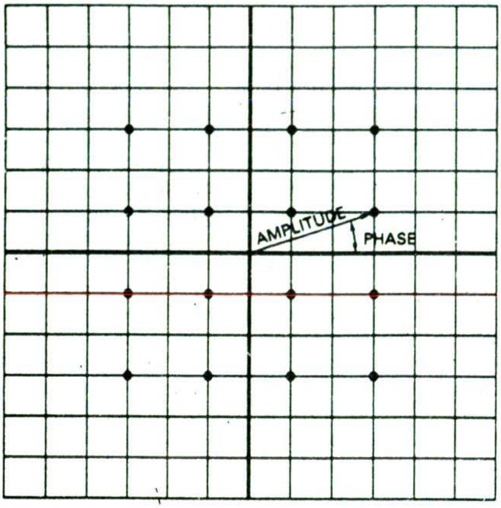


FIGURE 14-16 Phase-amplitude graph showing constellation for quaternary amplitude modulation (QAM).

a graph such as this is commonly referred to as a constellation.

Dibit and QAM phase-shift modulation permit higher data rates on phone lines, but correctly demodulating this type of phase-encoded data presents some unique problems. To illustrate the first problem, remember from our previous discussion that in a dibit system the value of a dibit is represented by shifting the phase of a carrier signal some specified number of degrees from a reference phase. In order to detect the amount of phase shift, the receiver and the transmitter must be using the same reference phase. This would be easy if we could just run another wire to carry a synchronizing clock signal. However, since this is not easily done, the synchronizing signal must in some way be included with the data. The carrier signal itself cannot be used directly, because that is the signal whose phase must be detected.

The solution to this problem is to use transitions in the transmitted signal to synchronize a phase-locked loop oscillator in the receiver. In order for this to work, two factors must be included in the transmitted data. First of all, the system must be operated synchronously rather than asynchronously, so that data, sync, or null characters are always being received by the receiver. Secondly, the transmitted data must have enough transitions at regular intervals to keep the phase-locked loop locked in the desired phase. The serial data stream from the USART may not have enough transitions in it to satisfy this second condition, so a special circuit called a *scrambler* is included in the transmitter part of the modem. The scrambler, which usually consists of a shift register with feedback, puts in extra signal transitions as needed. The output from the scrambler is then used to modulate the phase of the carrier. When the carrier signal reaches the receiver, the signal is demodulated to produce a signal of 1's and 0's. This signal is then passed through a descrambler, which reverses the scrambling process and outputs the original data.

A second problem encountered in high-speed data transmission with modems is error detection/correction. One method used to decrease the error rate is called *trellis coding*. Trellis coding uses a constellation with more points than the minimum required to represent the number of data bit combinations in the group. The information needed to decode each data bit is spread over several transmitted values rather than being encoded in just one as in straight QAM. This scheme makes it possible for the receiver to detect illegal values caused by errors. V.32-type modems use trellis coding to allow full-duplex 9600-bit/s transmission on a two-wire phone line with 2400-Bd modulation.

Note that this modulation rate is higher than we told you was possible for a phone line bandwidth of 2400 Hz. The actual bandwidth of the phone lines is usually 3000 or somewhat more, so it is common practice to "push" the bandwidth limits to get higher data transmission rates. Most modems are designed to work with several different transmission rates and modulation so that they can communicate with a variety of modems. The software controlling the modem usually attempts communication at the highest available data rate, and if the particular phone connection will not support that rate, it "falls back" to a lower data rate where it can successfully transmit and receive. V.32-type modems also contain echo cancellation circuitry to reduce errors caused by interference between the signal being sent out and the signal coming in.

Other techniques being used to increase the rate at which modems can transfer data on standard phone lines are error correcting and data compression. CCITT standard V.42 specifies an error detection/correction algorithm that is independent of the data transmission speed and modulation method. CCITT standard V.42 bis specifies data compression algorithms that can be implemented in modems independently of the data transmission rate and the modulation method. The algorithm in this standard allows up to a 4:1 data compression, depending on the amount of redundancy in the data being transmitted. An average increase of about 60% in the actual data transmission rate is common with this algorithm.

Still another technique used to increase the data rate on phone lines is Telebit Corporation's Dynamically Adaptive Multicarrier Quadrature Amplitude Modulation (DAMQAM). This scheme uses up to 512 different carrier frequencies within the bandwidth of the phone lines. Data transmission is spread out over a large number of these channels, so the transmission on any one channel can be a very low rate, even with an overall transmission rate of 19,200 bits/s.

Now that you know more than you may want to about the modulation schemes used in modems, let's take a look at how a high-speed modem can be interfaced with microcomputer buses.

MODEM HARDWARE OVERVIEW

Figure 14-17 shows a block diagram for a combination FAX and data modem which interfaces directly to the

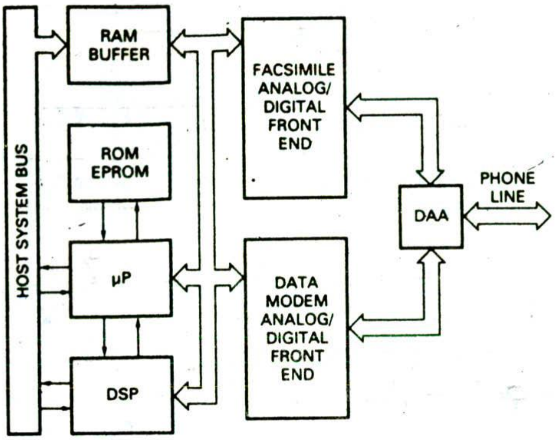

HOST SYSTEM BUS
RAM BUFFER
ROM EPROM
µP
DSP
FACSIMILE ANALOG/ DIGITAL FRONT END
DATA MODEM ANALOG/ DIGITAL FRONT END
DAA
PHONE LINE

FIGURE 14-17 Block diagram of combination FAX and data modem.

main buses in a microcomputer. As you can see, the modem contains a dedicated microprocessor to control the operations of the modem. This processor manages handshaking, data formatting, dialing, etc. The ROM or EPROM stores the program for the microprocessor and the RAM stores blocks of data received by the modem and blocks of data waiting to be sent.

As we described before, high-speed data transmission on phone lines requires precisely detecting the amplitude of signals, precisely detecting the phase of signals, noise filtering, and echo cancellation. In current modems these tasks are accomplished with the digital signal processing techniques we described in Chapter 10. As you can see, the modem in Figure 14-17 contains a dedicated digital signal processor to do all this.

The modem in Figure 14-17 contains a FAX front end and a data front end. The reason for this is that a FAX typically uses V.29 type half-duplex 9600-bit/s transmission, and the corresponding data communication uses V.22 bis full-duplex 2400-bit/s transmission.

The box labeled DAA in Figure 14-17 is the *data access arrangement* circuitry which actually interfaces the signals with the phone lines. This circuitry must conform to the provisions of FCC rules, Section 68.

LSI has made it possible to build a modem with very few parts. A device such as the Advanced Micro Devices AM7910, for example, can be used to produce a 1200-Bd FSK modem. The EXAR Corp. XR-2901 and 2902 chip set contains a major part of the circuitry needed to implement a modem which can send or receive facsimile (FAX) data at 9600 bits/s or send and receive full-duplex modem data at 2400 bits/s.

MODEM HANDSHAKING

Earlier in this chapter we gave an overview of the handshake process between a terminal and a remote computer through modems and the phone lines. Now that you know more about modems, we can take a closer look at the handshake sequence.

Most of the currently available modems contain a dedicated microprocessor. The built-in intelligence allows these units to automatically dial a specified number with either tones or pulses, and redial the number if it is busy or doesn't answer. When a smart modem makes contact with another modem, it will automatically try to set its transmit circuitry to match the baud rate of the other modem. Many modems can be set to automatically answer a call after a programmed number of rings so that you can access your computer from a remote location. Some units allow the user to establish a voice contact and then switch over to modem operation.

After a modem dials up another modem, a series of handshake signals takes place. The handshake signals may be generated by hardware in the modem or by software in the system connected to the modem. Figure 14-18, page 502, shows an example of the data and handshake waveforms for a modem built with the AM7910 single-chip FSK modem. Other modems may use a slightly different sequence, but the principles are the same.

The modem which makes a call is usually referred to as the *originate* modem, and the modem which receives the call is usually referred to as the *answer* modem. In the following discussion we will use the terms *calling modem* and *called modem*, respectively, to agree with the labels on the waveforms in Figure 14-18.

At the left side of the waveforms, a call is being made from one modem to another. Assuming that the $\overline{\text{DTR}}$ of the called modem is asserted, the ringing signal on the line will cause the DAA circuitry to assert the *ringing input* $\overline{\text{RI}}$ of the 7910. In response to this the 7910 will send out a silent period of about 2s to accommodate billing signals, and then it will send out an answer tone of 2025 Hz to the calling modem for 2 s. If the $\overline{\text{DTR}}$ and the $\overline{\text{RTS}}$ of the calling modem are asserted, indicating that data is ready to be sent, the calling modem then puts a tone of 2225 Hz (mark) on the line for 8 ms to let the called modem know that contact is complete. In response to this mark, the called modem asserts its *carrier-detect* output $\overline{\text{CD}}$ to enable the receiving UART. The calling modem then sends data until its $\overline{\text{RTS}}$ input is released by the computer or terminal sending the data. While it is receiving data on the main channel, the called modem can send data to the calling modem on the 5-bit/s back channel. Releasing $\overline{\text{RTS}}$ causes the modem to release $\overline{\text{CTS}}$ to the sending computer and remove the carrier from the line. The called modem senses the loss of the carrier and unasserts its carrier detect ($\overline{\text{CD}}$) signal.

Now, if the called system is to send some data back to the calling system on the main channel, it asserts the $\overline{\text{RTS}}$ input to its modem. The called modem sends a marking tone to the calling modem for 8 ms. The calling modem asserts its $\overline{\text{CD}}$ output to its UART. The called modem then sends data to the calling modem on the main channel until its $\overline{\text{RTS}}$ input is unasserted by the called system, indicating no more data to send. While the called modem is transmitting on the main channel, the calling modem can transmit over the back channel if necessary. The handshake is similar for a full-duplex system, but the data rates are equal in both directions.

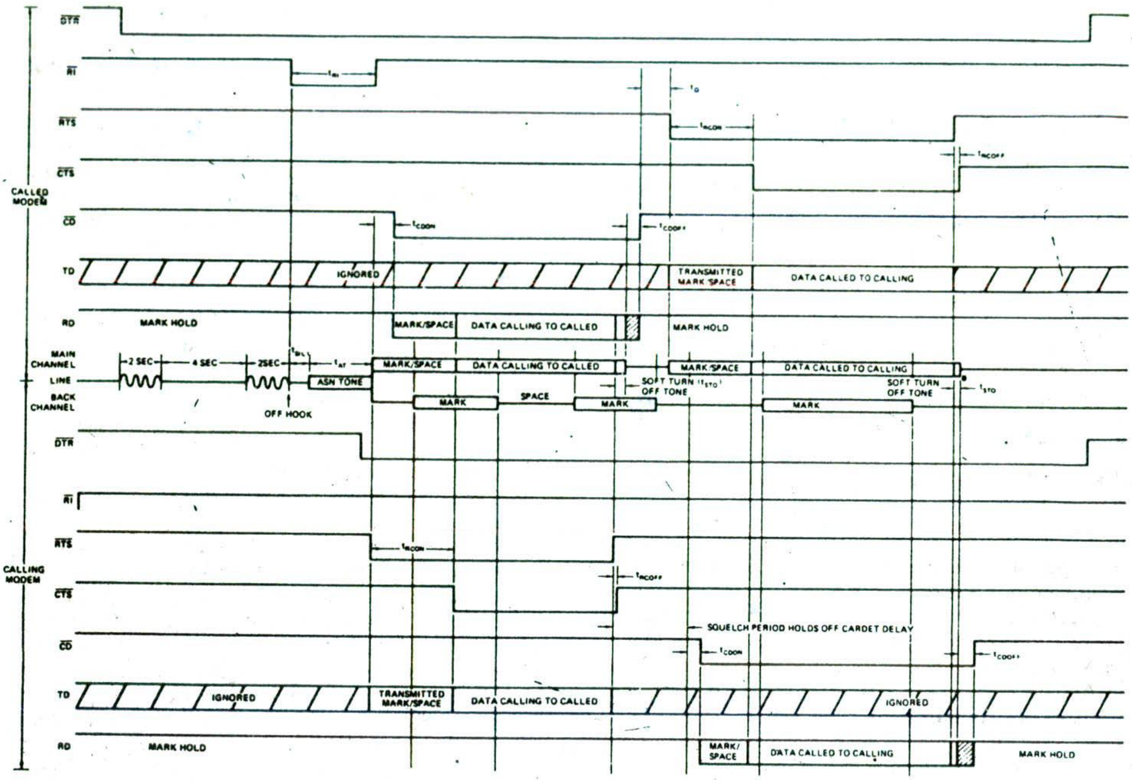


FIGURE 14-18 Handshake sequence for Bell-type 202 FSK modem using AM7910 modem chip. *(Copyright Advanced Micro Devices, Inc. (1982) Reprinted with permission of copyright owner. All rights reserved.)*

CODECs, PCM, TDM, and ISDN

In the previous sections we described how modems produce signals which are suitable for transmission over standard phone lines. Now we want to briefly discuss how telephone companies actually transmit the signals output by modems and some new developments which hopefully will eliminate the need for modems as we know them.

Digital signals have much better noise immunity than analog signals, so as soon as a phone company receives a voice or modem signal in its local branch office, the signal is converted to digital form. A D/A converter at the destination uses the received binary codes to reconstruct a replica of the original analog signal. Sending analog signals such as phone signals as a series of binary codes is called *pulse-code modulation* or PCM. The A/D converter that produces the binary codes in this application is usually called a *coder* and the D/A converter that reconstructs the analog signal from the pulse codes is referred to as a *decoder*. Since both a coder and a decoder are needed for two-way communication, they are often packaged in the same IC. This combined coder and decoder is called a *codec*. A common example of a codec is the Intel 2910A. This device contains a sample-and-hold circuit on the analog input, an 8-bit A/D converter, an 8-bit D/A converter, and appropriate control circuitry.

Normal A/D converters are linear, which means that the steps are the same size over the full range of the converters. The A/D converters used in codecs are nonlinear. They have small steps for small signals and large steps for large signals. In other words, for signals near the zero point of the A/D converter, it takes only a small change in the signal to change the code on the output of the A/D. For a signal near the full scale of the converter, a large change in the input signal is required to produce a change in the output binary code. This nonlinearity of the A/D converter is said to *compress* the signal, because it reduces the dynamic range of the signal. Compression in this way greatly improves the accuracy for small signals where it is needed, without going to a converter with more bits of resolution. The D/A in the codec is nonlinear in the reverse manner, so that when the binary pulse codes are converted to analog, the result is *expanded* to duplicate the original waveform. A codec which has this intentional nonlinearity is often referred to as a *compander* or a *companding codec*. Consult the Intel 2910A data sheet for more information about this.

In most systems the output of the codec A/D is not simply sent on a wire by itself, it is multiplexed with the outputs of many other codecs in a manner known as *time-division multiplexing* or TDM. There are several different formats used. A simple one will give you the idea of how it's done.

One of the first TDM systems was the T1 or DS-1 system, which multiplexes 24 PCM voice channels onto a single wire. For this system an 8-bit codec on each channel samples and digitizes the input signal at an 8-kHz rate. The 8-bit codes from the codecs are sent to a multiplexer which sends them out serially, one after the other. One set of bits from each of the 24 codecs plus a framing bit is referred to as a frame. Figure 14-19 shows the format of a frame for this system. The framing bit at the start of each frame toggles after each frame is sent. It is used to keep the receiver and the transmitter synchronized and for keeping track of how many frames have been sent. After it sends the framing bit, the multiplexer sends out the 8-bit code from the first codec, then sends out the 8-bit code from the next codec, and so on until the codes for all 24 have been sent out. At specified intervals the multiplexer sends out a frame which contains synchronization information and signaling information. This does not seriously affect the quality of the transmitted data.

Since the multiplexer is sending out 193-bit frames at a rate of 8000 per second, the data rate on the wire is 193 × 8000, or 1.544 Mbits/s. A newer system, known as T4M or DS-4, multiplexes 4032 channels onto a single coaxial cable or optic fiber. The bit rate for this system is 274.176 Mbits/s.

The question that should come to your mind about now is, If the phone company transmits data in high-speed digital form, why do I have to send data as modulated audio tones? The answer to this question is that the circuitry between your phone and the local branch office is a relic from a bygone analog era. This circuitry creates a "bottleneck" in the communications link.

One attempt to eliminate this bottleneck is a wideband digital connection system known as the *integrated services digital network* or ISDN. As shown in Figure 14-20*a*, page 504, ISDN replaces the analog connections between your home and the telephone company branch office with relatively high speed digital connections. An ISDN basic-rate service connection gives two 64-kbit/s voice/data channels and a 16-Kbit/s data/control channel in each direction. The voice/data channels are referred to as B1 and B2. The data/control channel is referred to as the D channel.

Figure 14-20*b* shows how the B1, B2, D, framing, and other bits are packed in a 48-bit frame for transmission.

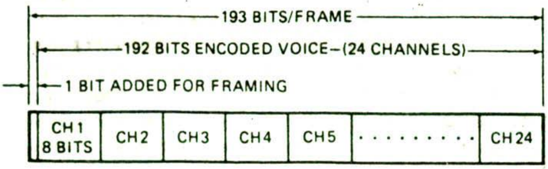


FIGURE 14-19 Frame format for telephone company T1 digital data transmission.

Note that a B channel has four times as many bits per frame as the D channel, so the bit rate for the B channel is four times the bit rate for the D channel. A 48-bit frame is transmitted every 250 μs, so the basic bit rate on the single line is 192 Kbits/s. Only 16 of the 48 data bits represent one of the B channels, so the transmission rate for a B channel is 64 Kbits/s.

For voice communication a codec in the telephone converts voice signals to a sequence of 8-bit codes which are then put in, for example, the B1 channel slots in the 48-bit frames. The codec also converts the codes for received voice signals back to analog form to drive a speaker. Some advantages of ISDN for standard telephone communications are that it gives better sound quality and allows identification of the number that a call is coming from.

For data communications an adapter in the computer formats the data to be transmitted in the required frames and adds the framing bits, etc. Since both B channels can be used, the effective data rate is the sum of that for the two channels, or 128K bits/s. In some cases the D channel can also be used for data, and the effective rate becomes the 144K bits/s.

In a large building with many telephones and computers, each phone and computer will communicate with the PBX in the building using a basic ISDN 2B + D service line such as we just described. The PBX will then use a higher-frequency multiplexed line such as the T1 system we described earlier to communicate with the telephone company's central office.

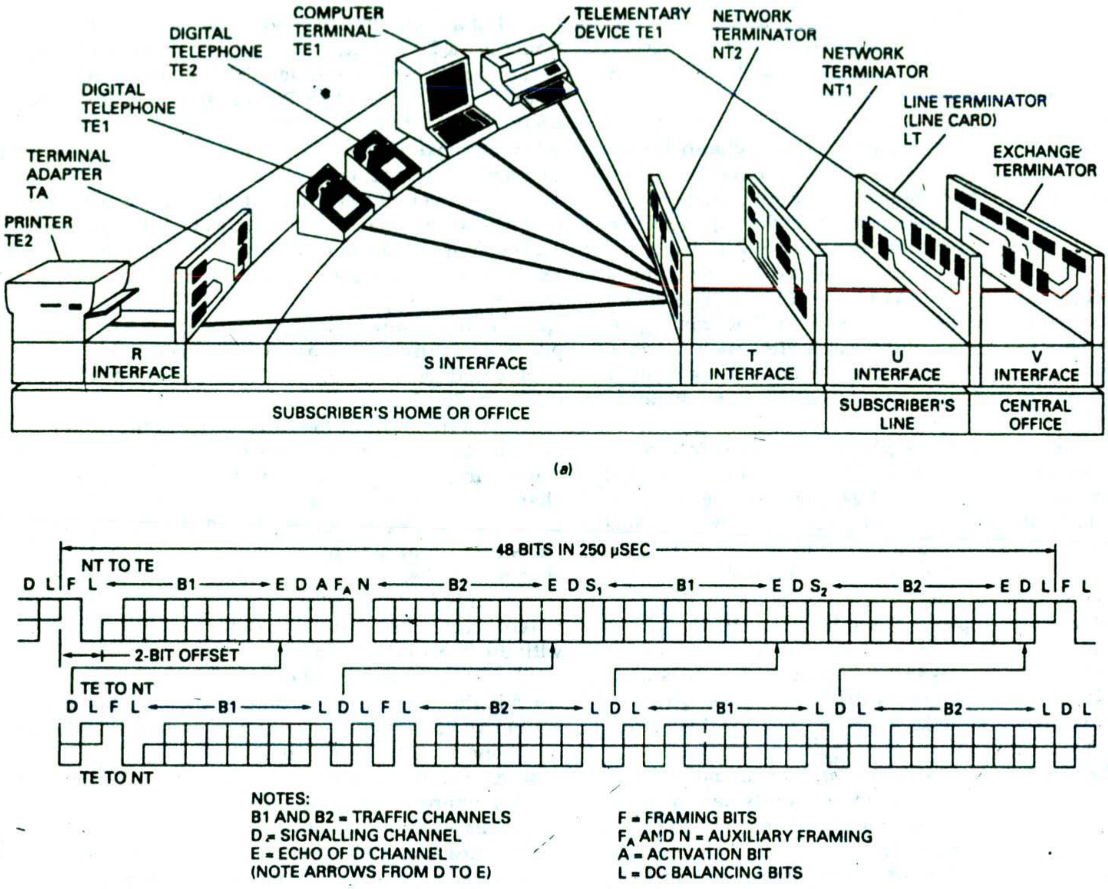
As you can see by the transmission rates for ISDN, it is a big improvement over the old analog connections. As of this writing ISDN is still available only in some major cities and a few other areas. It is slowly spreading to other areas, but if you want to communicate with many different locations, you will probably be stuck with an analog modem for some time. This is unfortunate, because high-speed data communication is required for interactive graphic user interfaces. In other words, if you want to rapidly transmit high-resolution color images, you need a high-speed communications link. In the next section we discuss fiber-optic systems, which allow the very high speed data transfer needed for this.

Fiber-Optic Data Communication

INTRODUCTION

All of the data communication methods we have discussed so far use metallic conductors. *Fiber-optic* systems use very thin glass or plastic fibers to transfer data as pulses of light. Some of the advantages of fiber-optic links are that they are immune to electrical noise, they can transfer data at very high rates, and they can transfer data over long distances without amplification.

Figure 14-21, page 505, shows the connections for a basic fiber-optic data link you can build and experiment with. This type of link might be used to transmit data from a sensor in an electrically noisy environment such as a factory. The light source here is a simple infrared LED. Higher-performance systems use an *infrared injection laser diode* (ILD) or some other laser driven by a high-speed, high-current driver. Digital data is sent over



(*a*)

(*b*)

FIGURE 14-20 Integrated services digital network (ISDN). (*a*) Line connections and interfaces. Reprinted from EDN, April 27, 1989, © 1989 CAHNERS PUBLISHING COMPANY, a Division of Reed Publishing USA. (*b*) Example S interface frame format showing how B1, B2, and D channel bits are packaged for transmission. NOTE: Frames sent from network termination to terminal equipment are offset 2 bits from frames sent from terminal equipment to network termination.

the fiber by turning the light beam on for a 1 and off for a 0.

> NOTE: If you are working on a fiber-optic system you should never look directly into the end of the fiber to see if the light source is working, because the light beam from some laser diodes is powerful enough to cause permanent eye damage. Use a light meter, or point the cable at a nonreflective surface to see if the light source is working.

To convert the light signal back into an electrical signal at the receiving end, Darlington photodetectors such as the MFOD73 shown in Figure 14-21, PIN FET devices, or avalanche photodiodes (APDs) are used. APDs are more sensitive and operate at higher frequencies, but the circuitry for them is more complex. A Schmitt trigger is usually used on the output of the detector to "square up" the output pulses.

The fiber used in a cable is made of special plastic or glass. Fiber diameters used range from 2 to 1000 μm. Larger-diameter plastic fibers are used for short-distance, low-speed transmission, and small-diameter glass fibers are used for high-speed applications such as long-distance telephone transmission lines. As shown in Figure 14-22*e*, page 506, the fiber-optic cable consists of three parts. The optical-fiber core is surrounded by a *cladding* material which is also transparent to light. An outer sheath protects the cladding and prevents external light from entering.

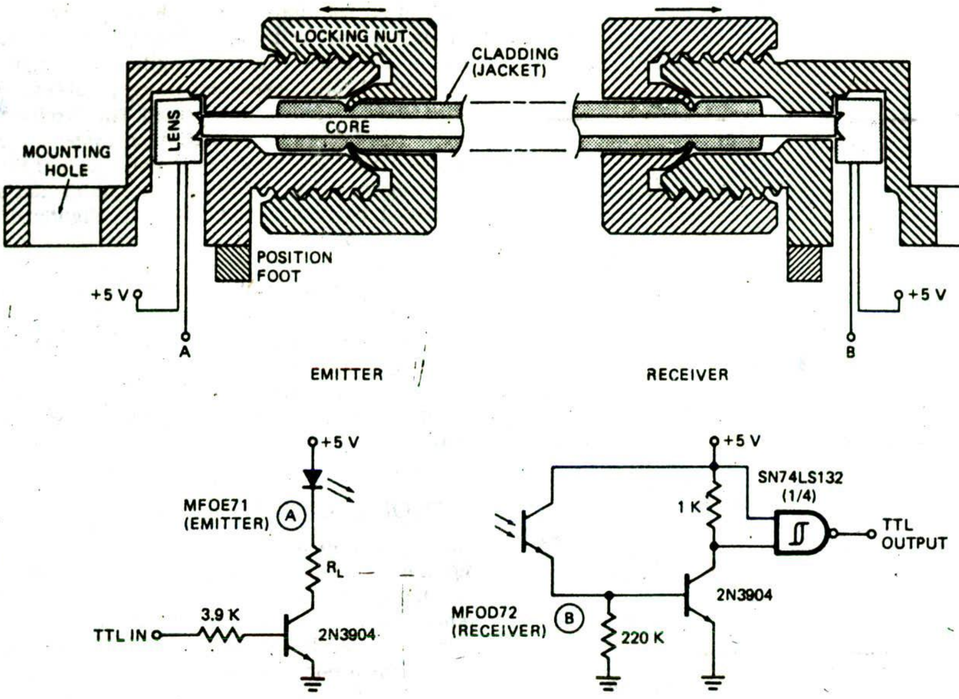


FIGURE 14-21 Components of a simple fiber-optic data link.

Now that you have an overview of an optical-fiber link, let's take a look at how the light actually propagates through the fiber and the trade-offs with different fibers.

THE OPTICS OF FIBERS

The path of a beam of light going from a material with one optical density to a material of different optical density depends on the angle at which the beam hits the boundary between the two materials. Figure 14-22 shows the path that will be taken by beams of light at various angles going from an opticaly dense material such as glass to a less dense material such as a vacuum or air. If the beam hits the boundary at the right angle, it will go straight through, as shown in Figure 14-22*a*. When a beam hits the boundary at a small angle away from the perpendicular or *normal*, it will be bent away from the normal when it goes from the more dense to the less dense, as shown in Figure 14-22*b*. A light beam going in the other direction would follow the same path. A quantity called the *index of refraction* is used to describe the amount that the light beam will be bent. Using the angle identifications shown in Figure 14-22*b*, the index of refraction, *n*, is defined as the (sine of angle *B*)/(sine of angle *A*). A typical value for the index of refraction of glass is 1.5. The larger the index of refraction, the more the beam will be bent when it goes from one material to another.

Figure 14-22*c* shows a unique situation that occurs when a beam going from a dense material to a less dense material hits the boundary at a special angle called the *critical angle*. The beam will be bent so that it travels parallel to the boundary after it enters the less dense material.

A still more interesting situation is shown in Figure 14-22*d*. If the beam hits the boundary at an angle greater than the critical angle, it will be totally reflected from the boundary at the same angle on the other side of the normal. This is somewhat like skipping stones across water. In this case the light beam will not leave the more dense material.

To see how all this relates to optical fibers, take a look at the cross-sectional drawing of an optical fiber in Figure 14-22*e*. If a beam of light enters the fiber parallel to the axis of the fiber, it will simply travel through the fiber. If the beam enters the fiber so that it hits the glass-cladding layer boundary at the critical angle, it will travel through the fiber-optic cable in the cladding layer as shown for beam *Y* in Figure 14-22*e*. However, if the beam enters the cable so that it hits the glass-cladding layer boundary at an angle greater than the critical angle, it will bounce back and forth between the walls of the fiber as shown for beam *X* in Figure 14-22*e*. The glass or plastic used for fiber-optic cables has very low absorption, so the beam can bounce back and forth along the fiber for several feet or miles without excessive attenuation. Most systems use light with wavelengths of 0.85μm, 1.3 μm, or 1.500 μm, because absorption of light by the optical fibers is minimum at these wavelengths.

If an optical fiber has a diameter many times larger than the wavelength of the light being used, then beams

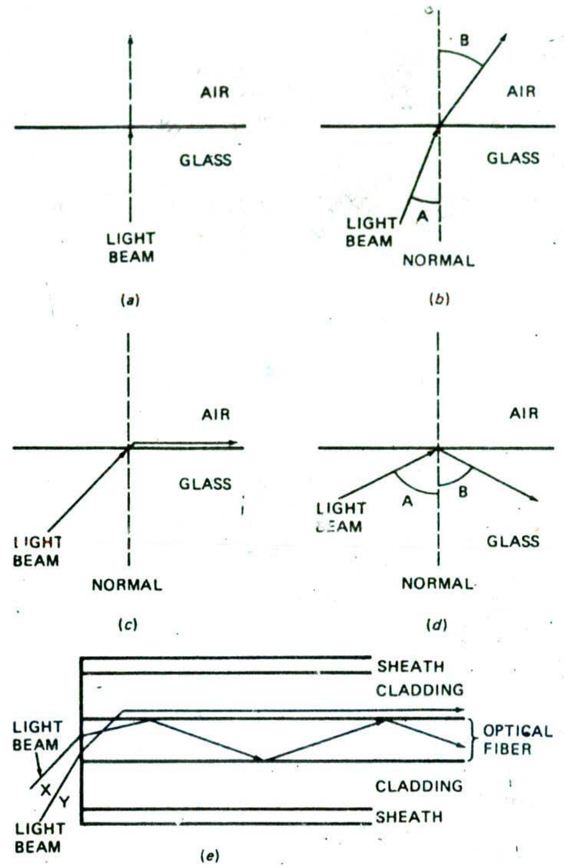


FIGURE 14-22 Light-beam paths for different angles of incidence with the boundary between higher optical density and lower optical density materials. (*a*) Right angle. (*b*) Angle less than critical angle. (*c*) At critical angle. (*d*) At angle greater than critical angle. (*e*) At angle greater than critical angle in optical fiber.

which enter the fiber at different angles will arrive at the other end of the fiber at slightly different times. The different angles of entry for the beams are referred to as *modes*. A fiber with a diameter large enough to allow beams with several different entry angles to propagate through it is called a *multimode* fiber. Since multimode fibers are larger, they are easier to manufacture, are easier to manually work with, and can use inexpensive LED drivers. However, the phase difference between the output beams in multimode fibers causes problems at high data rates. One partial solution to this problem is to dope the glass of the fiber so that the index of refraction decreases toward the outside of the fiber. Light beams travel faster in the region where the index of refraction is lower, so beams which take a longer path back and forth through the faster outer regions tend to arrive at the end of the fiber at the same time as those that take a shorter path through the slower center region.

A better solution to the phase problems of the multimode fiber is to use a fiber that has a diameter only a few times the wavelength of the light being transmitted. Only beams very nearly parallel to the axis of the fiber can then be transmitted. This is referred to as *single-mode* operation. Single-mode systems currently available can transmit data a distance of over 60 km at a rate greater than 1 Gbit/s. An experimental system developed by AT&T multiplexes 10 slightly different wavelength laser beams onto one single-mode fiber. The system can transmit data at an effective rate of 20 Gbit/s over a distance of 68 km without amplification.

One of the main problems with single-mode fibers is the difficulty in making low-loss connections with the tiny fibers. Another difficulty is that in order to get enough light energy into the tiny fiber, relatively expensive laser diodes or other lasers rather than inexpensive LEDs must be used.

FIBER-OPTIC CABLE USES

Fiber-optic transmission has the advantages that the signal lines are much smaller than the equivalent electrical signal lines, signals can be sent much longer distances without repeater amplifiers, and very high data rates are possible. One of the first major uses of fiber-optic transmission systems has been for carrying large numbers of phone conversations across oceans and between cities. A single 12-fiber, ½-in. diameter optical-fiber cable can transmit 1,000,000 simultaneous telephone conversations. These specifications are impressive, but relatively primitive as compared to the possibilities shown by laboratory research. In the future it is possible that the high data rate of fiber-optic transmission may make picture phones a household reality, replace TV cables, replace satellite communication for many applications, replace modems, and provide extensive computer networking.

ASYNCHRONOUS COMMUNICATION SOFTWARE ON THE IBM PC

In a previous section of this chapter we discussed how asynchronous serial data can be sent or received with an 8251A on a polled or an interrupt basis. Any serial communication at some point has to get down to that level of hardware interaction. However, as we tried to show you in Chapter 13, you should write programs at the highest language level you have available without excessively sacrificing execution speed, the amount of memory used, or ease of use. In this section we show examples of serial data communication using direct UART interaction, BIOS function calls, DOS function calls, and C function calls, so that you can write programs at any level.

As a first example we will show you how we developed a simple terminal emulator program; then as a second example we will show you how we developed a program which downloads object code files from the IBM PC to an SDK-86 board.

A Terminal Emulator Using DOS Function Calls and BIOS Calls

As a first step in developing the SDK download program, we decided to write a simple terminal emulator program. A terminal emulator program, when run, makes the PC act like a dumb CRT terminal. Characters typed on the keyboard are sent out on an RS-232C line to a modem or some other RS-232C-compatible equipment, and characters coming into the PC on an RS-232C line are displayed on the CRT.

Whenever you want to write a system program such as this, you should first write the algorithm in a standard form as we taught you in Chapter 3. Figure 14-23 shows an algorithm for our simple terminal emulator.

The next step is to decide what language you want to work in and translate the algorithm to that language. For this example we decided to use DOS function calls and BIOS function calls. If you are programming at this level you should first see what DOS function calls are available to do each part of the algorithm for you. There are several reasons for this approach. First, DOS function calls are usually very easy to use because they do not require you to have extensive knowledge of the hardware details. Second, programs written at the DOS level are much more likely to run correctly on another "compatible" system. If you are going to be writing system programs for the IBM PC or PS/2, you should get a copy of the *DOS Technical Reference Manual*.

If you need some operation that is not provided by a DOS function call, then the next step is to check the available BIOS procedures in the IBM PC or PS/2 *Technical Reference Manual*. Finally, if neither DOS or BIOS has the functions you need, you invoke the 5-minute rule, and then dig into the Technical Reference Manual to find the pieces you need to write the functions yourself.

The relevant DOS function calls are as follows.

Function Call 2—The character in DL is sent to the CRT and the cursor is advanced one position.

Function Call 3—Waits for a character to be received in the serial port and then returns with character in AL.

Function Call 4—The character in DL is output to the serial port.

Function Call 8—Waits for a key to be pressed on the keyboard and then returns the ASCII code for the key in AL.

```
INIT COM1
REPEAT
    IF CHARACTER READY IN UART THEN
        READ CHARACTER
        SEND TO CRT
    IF KEYPRESSED ON IBM KEYBOARD THEN
        READ KEY
        SEND TO SERIAL PORT
UNTIL FOREVER
```

FIGURE 14-23 Algorithm for simple terminal emulator program.

Function call 2 looks useful for sending a character to the CRT, and function call 4 looks useful for sending a character to the serial port. However, the keyboard call and the serial-read call will not work because they both sit in loops waiting for input. In other words, if you call function 8, execution will not return from that function until a key is pressed on the keyboard. If execution is in the function 8 loop, characters coming into the serial port will not be read. What is needed here are procedures which allow polling to go back and forth between the keyboard and the serial port receiver. Also needed is the least painful way to initialize the serial port. Let's see what BIOS has to offer.

Figure 14-24 shows the subfunctions and parameter passing registers for the IBM PC BIOS INT 14H procedure. This procedure will do one of four functions,

```
BIOS INT 14H SUBPROCEDURES AND PARAMETERS

AH       FUNCTION

0     Initialize communications port
      DX = port number - 0 = com1, 1 = com2
      AL = comm port mode word as follows.
      baud rate     parity   stop   data bits
      7 6 5          4 3      2       1 0
      0 0 0 110    X0-none   0-1    10-7
      0 0 1 150    01-odd    1-2    11-8
      0 1 0 300
      0 1 1 600
      1 0 0 1200
      1 0 1 2400
      1 1 0 4800
      1 1 1 9600

1     Send the character in AL out comm port
      DX = port number - 0 = com1, 1 = com2
      Returns line status in AH as shown
      below for AH = 3 call.

2     Read character from com port
      DX = port number - 0 = com1, 1 = com2
      Returns character in AL, status in AH.
      If AH bit 7 = 1, unable to read char.
      If AH bit 7 = 0, bits 3,2,1 flag errors.
      If AH = 0, char read with no errors.

3     Read line and modem status.
      DX = port number - 0 = com1, 1 = com2
      Returns: AH = line status
        bit 7 = time out
        bit 6 = transmit shift register empty
        bit 5 = transmit hold register empty
        bit 4 = break detect
        bit 3 = framing error
        bit 2 = parity error
        bit 1 = overrun error
        bit 0 = received character ready
               AL = modem signal status
        bit 7 = receive line signal detect
        bit 6 = ring indicator
        bit 5 = data set ready asserted
        bit 4 = clear to send asserted
        bit 3 = delta receive line detect
        bit 2 = trailing edge ring detect
        bit 1 = delta data set ready
        bit 0 = delta clear to send
```

FIGURE 14-24 Subprocedures and parameter passing registers for IBM PC BIOS INT 14H procedure.

depending on the value passed to it in the AH register. If AH = 0 when the procedure is called, the byte in AL is used to initialize the serial port device as shown. If AH = 1, then the character in AL will be sent out from the serial port. Likewise, if AH = 2, then a character will be read in from the serial port and left in AL. Finally, if AH = 3 when the procedure is called, the status of the serial port will be returned in AH and AL. The first of these four options solves the initialization problem. The last (AH = 3) supplies most of the solution for the problem of determining when the UART has a character ready to be read. Bit 0 of the status byte returned in AH will be set if the UART contains a character ready to be read. If a character is ready, it can be read in and sent to the CRT. If no character is present, the program can go check to see if a key on the keyboard has been pressed.

Figure 13-1 shows the subprocedures and parameter-passing registers for the IBM PC BIOS INT 16H procedure which accesses the keyboard. Remember from the discussion at the start of the last chapter that this procedure performs one of three different functions, depending on the value passed to it in AH. If AH = 0, the procedure will wait for a keypress and return the code for the pressed key in AL or AH. If AH = 1, the function will return with the zero flag = 0 if a key has been pressed and the code is available to be read. If AH = 2, the shift status will be returned in AL.

Calling the INT 16 procedure with AH = 1 solves the problem of polling the keyboard without sitting in a loop the way the DOS function call does. The zero flag can simply be checked upon return from the INT 16 procedure, and if no key is ready, execution can go check the UART again. If a key code is ready, it can be read in with a DOS call or another INT 16 call and sent to the UART.

Figure 14-25 shows a simple terminal emulator program which uses the procedures we have described. The program follows the algorithm almost line by line. Remember from previous chapter examples that BIOS procedures are called directly by an INT (number) instruction, and DOS calls are done by putting the function number in AH and doing an INT 21H instruction.

You can connect the serial port on an IBM PC- or PS/2-compatible to the serial port of an SDK-86 board as shown in Figure 14-10*b* and use this program to communicate with the board at 300 or 600 Bd. However, if you try to use the program at 1200 or 2400 Bd, the first character of each line of characters received from the SDK-86 or other source will be lost. It took some work to figure out why this is the case, because even with a 4.77-MHz 8088, the computer should be more than fast enough to handle 4800-Bd communciation with no trouble. The problem is in the INT 10H procedure which we used to send characters to the CRT. After a carriage return is sent to the CRT, the display on the screen is scrolled up one line. To avoid flicker, however, the screen is not scrolled until the next frame update. Since the frame rate for the monochrome display is 50 Hz, the return from the display procedure may take as long as 20 ms. Any characters that come into the UART during this time will be lost. The next section shows how we solved this problem to produce a download program which works correctly at 4800 baud.

IBM PC to SDK-86 Download Program

The main purpose of the program described in this section is to allow the binary codes for programs developed on an IBM PC- or PS/2-compatible computer to be downloaded through an RS-232C link to an SDK-86 board. The program also functions as a dumb terminal so that downloaded programs can be run, memory contents displayed, and registers examined by using the SDK-86 serial monitor commands. These commands are implemented by simply typing the appropriate keys on the computer keyboard.

Figure 14-26, page 510, shows the overall algorithm for the program. The main difference between this algorithm and the one for the dumb terminal in Figure 14-24 is the addition of the actions when the letter Q or the letter L is typed. However, we implemented the algorithm in a different way in order to solve the speed problem described in the previous section, to give you some more exposure to the C programming language, and to show you some very important programming techniques. Incidentally, the 1986 version of this program, written entirely in assembly language, required six pages. This version, written mostly in C, requires only four pages. It would be shorter still except that we left two procedures in assembly language so you can more easily work your way through them if you want.

Figure 14-27, page 511, shows the complete program. The main part of the program is only a little over a page long. Main calls six functions: INIT, SERIAL_IN, CHK_N_DISPLAY, xmit, convert_and_send, and shutdown to do most of the work. After we give an overview, we will explain in detail how each of these functions work.

OVERVIEW

As you can see from the algorithm in Figure 14-26, this program spends most of the time running around a loop which waits for the user to press a key or a character to be received in a buffer from the SDK-86. In our program the two statements while (!bioskey(1)) and CHK_N_DISPLAY () implement this loop.

The bioskey (1) part of this calls the BIOS INT 16H procedure to determine if a key has been pressed. If a key has been pressed, we call bioskey again to read in the code for the pressed key and use a switch structure to determine the action to take for that key code. If no key has been pressed, the bioskey (1) function call returns a 0 and the while loop repeats.

When the CHK_N_DISPLAY function is called, it determines if a buffer contains any characters read in from the UART connected to the SDK-86. If the buffer contains no characters, execution simply returns to the while loop. If the buffer contains a character, the character is sent to the CRT. Here's how this program solves the timing problem suffered by the program in Figure 14-25.

Remember from the discussion in the previous section

```
;8086 PROGRAM F14-25.ASM
;TERMINAL EMULATOR PROGRAM FOR SDK-86
;  This program sends characters entered on the IBM PC to the COM1
;  serial port at 600 baud, and displays characters received from the
;  COM1 serial port on the CRT.
PAGE,132

STACK_HERE      SEGMENT     STACK
                DW 100 DUP(0)
                STACK_TOP LABEL WORD
STACK_HERE      ENDS

CODE_HERE      SEGMENT
     ASSUME CS:CODE_HERE, SS:STACK_HERE

START: MOV  AX, STACK_HERE       ; Initialize stack segment
       MOV  SS, AX
       MOV  SP, OFFSET STACK_TOP ; Initialize stack pointer
       MOV  AH, 00               ; Initialize COM1
       MOV  DX, 0000             ; Point at COM1
       MOV  AL, 01100111B        ; 600 Bd, no parity, 2 stop,8-bit
       INT  14H                  ; via BIOS INT 14H
       STI                       ; Enable interrupts
CHKAGN:MOV  DX, 0000             ; Point at COM1
       MOV  AH, 03               ; Check for character from SDK
       INT  14H
       TEST AH, 01H              ; See if char waiting in UART
       JNZ  RDCHAR               ; If char, read it
       JMP  KYBD                 ; else, go look for keypress
RDCHAR:MOV  AH, 02               ; Read character
       INT  14H                  ; from UART to AL
       MOV  DL, AL               ; Character to DL for DOS call
       MOV  AH, 02H              ; DOS call number for CRT display
       INT  21H                  ; Do DOS call
KYBD:  MOV  AH, 1                ; Check if key has been pressed
       INT  16H                  ;  using BIOS call
       JNZ  RDKY                 ; If keypress, read key code
       JMP  CHKAGN               ; else look for more from SDK
RDKY:  MOV  AH, 0                ; Read key code
       INT  16H                  ; using BIOS call
       MOV  DX, 0000H            ; Point at COM1 serial port
       MOV  AH, 01
       INT  14H                  ; Send character to UART with BIOS
       JMP  CHKAGN               ; Go look for another char from UART
                                 ; or from keyboard
CODE_HERE   ENDS
       END START
```

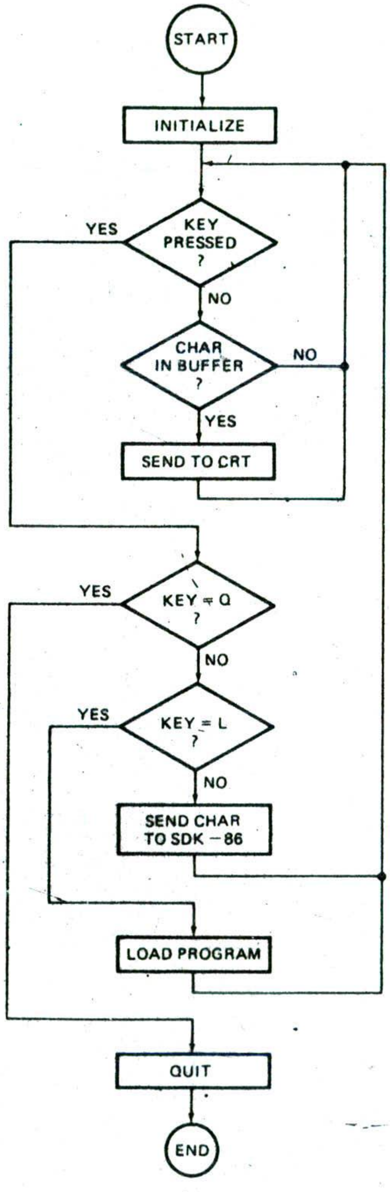
FIGURE 14-25 Simple terminal emulator program using DOS and BIOS function calls.

that if the program in Figure 14-25 is operated at over 600 Bd, characters which come into the UART while the INT 10H procedure is scrolling the CRT display are missed. In this program we read characters from the UART on an interrupt basis and put them in a buffer. Even if the PC is in the middle of the INT 10H procedure or some other procedure when the UART has a character ready, the interrupt procedure will read the character from the UART and put it in the buffer. When execution loops back around to the CHK_N_DISPLAY procedure again, the character will be read from the buffer and sent to the display. SERIAL_IN is the interrupt procedure which reads characters from the UART and puts them in the buffer. Now let's take a closer look at the switch structure which executes when the user presses a key on the PC keyboard.

If the user presses a Q key, we call the shutdown function to put the system back in its initial state and exit to DOS. If the user presses an L key, we first prompt the user for the path and name of a .BIN or .COM file to be downloaded to the SDK-86. After reading in the name of the file, we open the file if possible and send the

```
INITIALIZE EVERYTHING
   REPEAT
      IF KEY PRESSED THEN
         READ KEY
         IF KEY = Q THEN
              QUIT
         ELSE IF KEY = L THEN
                DOWNLOAD BINARY FILE FROM DISK TO SDK-86
              ELSE SEND CHARACTER FOR PRESSED KEY TO SDK-86
      IF UART BUFFER HAS CHARACTER THEN
         SEND CHARACTER TO CRT
   UNTIL QUIT
```

(a)



(b)

FIGURE 14-26 Algorithm for SDKCOM1 download program. (*a*, top) Pseudocode. (*b*, left) Flowchart.

Substitute command to the SDK-86. The while loop starting with while (!feof(fp)) reads a data byte from the file stream and then calls the convert_and_send function to convert the data byte to the format required by the SDK-86 and send it to the UART for transmission to the SDK-86. We then load numchar with the number of characters and call the CHK_N_DISPLAY procedure to send the SDK-86 response to the CRT.

Note that we have to use the feof () function to detect the end of this file rather than using the EOF character method we showed you for text files in the last chapter. The reason for this is that in a binary file such as a .BIN or .COM file, the EOF character, FFH, is a valid data byte and cannot be used as an end-of-file marker.

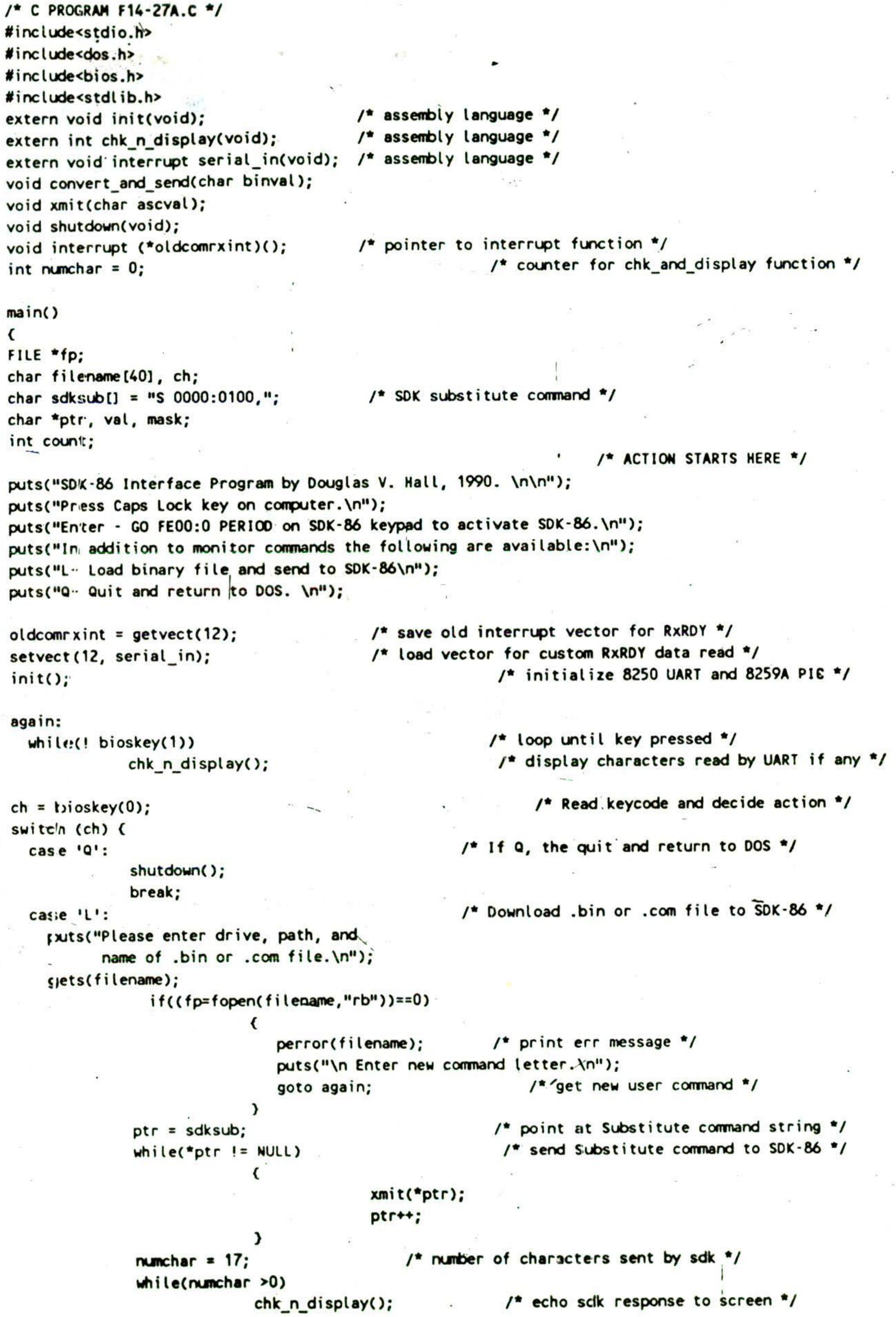
After all the data bytes are sent, we use the xmit function to send a carriage return to the SDK-86. This executes the SDK-86 Substitute command. Finally, we close the file before breaking from the switch structure.

If the key pressed by the user is not a Q or an L, the default part of the switch structure calls the xmit function to simply send the code for the pressed key on to the SDK-86.

The goto again; statement after the switch statement sends execution back to the while loop, which waits for a keypress or a character received in the buffer. Now let's dig a little deeper into the major parts of the program.

INITIALIZATION

The UART used on the IBM asynchronous communications adapter board is an INS8250. If the board is configured as system serial port COM1, the interrupt output from this device is connected to the IR4 input of an 8259A priority-interrupt controller on the main PC board of the IBM computer. The major part of the initialization here involves getting the 8250 initialized and setting up the interrupt mechanism. Remember from previous discussions that when an 8259A receives an interrupt on an IR input that is unmasked, it sends a specified interrupt type to the processor. The 8259A is initialized by the BIOS so that type 8 will be sent for an IR0 input. Therefore, for an IR4 signal from the UART, the 8259A will send type 0CH to the processor.



```
/* C PROGRAM F14-27A.C */
#include<stdio.h>
#include<dos.h>
#include<bios.h>
#include<stdlib.h>
extern void init(void);                 /* assembly language */
extern int chk_n_display(void);         /* assembly language */
extern void interrupt serial_in(void);  /* assembly language */
void convert_and_send(char binval);
void xmit(char ascval);
void shutdown(void);
void interrupt (*oldcomrxint)();        /* pointer to interrupt function */
int numchar = 0;                                    /* counter for chk_and_display function */

main()
{
FILE *fp;
char filename[40], ch;
char sdksub[] = "S 0000:0100,";          /* SDK substitute command */
char *ptr, val, mask;
int count;
                                                         /* ACTION STARTS HERE */
puts("SDK-86 Interface Program by Douglas V. Hall, 1990. \n\n");
puts("Press Caps Lock key on computer.\n");
puts("Enter - GO FE00:0 PERIOD on SDK-86 keypad to activate SDK-86.\n");
puts("In addition to monitor commands the following are available:\n");
puts("L- Load binary file and send to SDK-86\n");
puts("Q- Quit and return to DOS. \n");

oldcomrxint = getvect(12);              /* save old interrupt vector for RxRDY */
setvect(12, serial_in);                 /* load vector for custom RxRDY data read */
init();                                             /* initialize 8250 UART and 8259A PIC */

again:
  while(! bioskey(1))                               /* loop until key pressed */
            chk_n_display();                         /* display characters read by UART if any */

ch = bioskey(0);                                        /* Read keycode and decide action */
switch (ch) {
  case 'Q':                                       /* If Q, the quit and return to DOS */
            shutdown();
            break;
  case 'L':                                       /* Download .bin or .com file to SDK-86 */
    puts("Please enter drive, path, and
          name of .bin or .com file.\n");
    gets(filename);
              if((fp=fopen(filename,"rb"))==0)
                    {
                      perror(filename);        /* print err message */
                      puts("\n Enter new command letter.\n");
                      goto again;                   /* get new user command */
                    }
            ptr = sdksub;                           /* point at Substitute command string */
            while(*ptr != NULL)                      /* send Substitute command to SDK-86 */
                    {
                              xmit(*ptr);
                              ptr++;
                    }
            numchar = 17;                /* number of characters sent by sdk */
            while(numchar >0)
                    chk_n_display();           /* echo sdk response to screen */
```

FIGURE 14-27 C and assembly language source program for SDKCOM1 program. (a) C mainline. (b) Assembly language modules. (*Continued on pp. 512–14.*)

```
            while(!feof(fp))                              /* process bytes until end of file */
                    {
                    val=getc(fp);                 /* read character from file stream */
                    if(!feof(fp))
                      {
                      convert_and_send(val);/* ASCII values to SDK-86 */
                      numchar = 14;              /* characters returned by SDK-86 */
                      while(numchar >0)
                        chk_n_display();         /* send sdk echo in */
                                                        /*  ring buffer to CRT */
                      }
                    }
            val = 0x0d;                           /* send terminator character to SDK-86 */
            xmit(val);
            fclose(fp);                           /* close file */
            break;                                /* go get next user command */
    default:                                      /* not Q or L, just send to SDK-86 */
            xmit(ch);
            break;
    }
    goto again;
  }                                                        /* end of main */

  void convert_and_send(char binval)
  {
    char hold;
    hold = binval;                                 /* Save copy of binary value */
    binval = binval >>4;                       /* upper four bits to lower */
    binval = binval & 0x0f;                  /*mask upper four bits */
    if(binval >= 0x0A)                             /* convert nibble to ASCII */
              binval = binval + 0x37;        /* add 37H if letter */
    else
              binval = binval + 0x30;        /* add 30H if number */
    xmit(binval);                                          /* send to SDK-86 */
    binval = hold;                                        /* get original value of binval */
    binval = binval & 0x000f;                      /* mask upper nibble */
    if(binval >= 0x0A)                                /* convert lower nibble to ASCII */
              binval = binval + 0x37;        /* add 37H if hex letter */
    else
              binval = binval + 0x30;        /* add 30H if hex number */
    xmit(binval);
    binval = 0x2c;                                            /* load code for comma */
    xmit(binval);
  }                                                             /* send to SDK-86 */

  void xmit(char ascval)
  {
    while((bioscom(3,0,0)& 0x2000)==0)
           ;                                                        /* wait for UART xmit buffer ready */
    outportb(0x3f8, ascval);              /* send ASCII character to UART */
  }

  void shutdown(void)
  {
                                                                    /* shut everything down and return to DOS */
    unsigned char mask;
    mask = inportb(0x21);                 /* mask 8259 IR4 interrupt input */
    mask = mask | 0x1C;
    outportb(0x21,mask);
    setvect(12, oldcomrxint);                        /* restore BIOS interrupt vector */
    exit(0);                                                 /* automatically closes any open files */
  }
```

FIGURE 14-27 (*Continued*)

```
;8086 PROGRAM F14-27B.ASM
_TEXT      SEGMENT     BYTE PUBLIC 'CODE'
     DGROUP      GROUP      _DATA
     ASSUME      CS:_TEXT,DS:DGROUP,SS:DGROUP
_TEXT      ENDS

_DATA      SEGMENT WORD PUBLIC 'DATA'
     QUEUE DB 1000 DUP(0)             ; Declare ring buffer
     HEAD_POINTER DW 0                ; Pointer to read location in buffer
     TAIL_POINTER DW 0                ; Pointer to write location in buffer
     TIME_OUT_MESS DB 'TRANSMIT TIMEOUT - CHECK HARDWARE',0DH, 0AH
     EXTRB     _NUMCHAR:WORD
_DATA      ENDS

_TEXT      SEGMENT     BYTE PUBLIC 'CODE'
     PUBLIC _INIT
     PUBLIC _SERIAL_IN
     PUBLIC _CHK_N_DISPLAY

   _INIT PROC NEAR
                              ; Unmask 8259A IR4
     IN   AL, 21H             ; Read 8259A IMR
     AND  AL, 0ECH            ; Unmask IR4
     OUT  21H, AL
;Initialize 8250 UART baud rate,etc.
     MOV  AH, 00              ; Initialize COM1
     MOV  DX, 0000            ; Point at COM1
     MOV  AL, 11000111B       ; 4800 Bd,No parity,2 stop,8-bit
     INT  14H                 ;  via BIOS INT 14H
;Enable 8250 RxRDY interrupt
     MOV  DX, 03FBH           ; Point at 8250 line control port
     IN   AL, DX              ; Read in line control word
     AND  AL, 7FH             ; Set DLAB = 0
     OUT  DX, AL              ; Send line control word back out
     MOV  AL, 01H             ; Value to enable RxRDY interrupt
     MOV  DX, 03F9H           ; Point at interrupt enable register
     OUT  DX, AL              ; Enable RxRDY interrupt
     MOV  AL, 0BH             ; Assert 8250 OUT2, RTS, DTR byte
     MOV  DX, 03FCH           ; Point at modem control reg in 8250
     OUT  DX, AL              ; Send to 8250
     RET
  _INIT ENDP

   _SERIAL_IN PROC FAR
     STI                      ;Interrupts back on for clock, etc.
     PUSH AX
     PUSH BX
     PUSH DX
     PUSH DI
     PUSH DS
     MOV AX, DGROUP           ; Load current DS register value
     MOV DS, AX
     MOV  DX, 03F8H           ; Receiver buffer address for 8250
     IN   AL, DX              ; Read character
     MOV  DI, TAIL_POINTER    ; Get current tail pointer value
     INC  DI                  ; Point to next storage location
     CMP  DI, 1000            ; Compare with max to see if time
                              ;  to wrap around
     JNE  FULCHK              ; No, go check if queue full
     MOV  DI, 00              ; Yes, set DI for wraparound to start
```

FIGURE 14-27 (*Continued*)

```
FULCHK:
     CMP  DI, HEAD_POINTER    ; Check for full queue
     JE   NO_MORE             ; Full, do not write char
     MOV  BX, TAIL_POINTER    ; Not full, point at write loc
     MOV  QUEUE[BX], AL       ; Character to circular buffer
     MOV  TAIL_POINTER, DI    ; Save new tail pointer value
NO_MORE:

     MOV  AL, 20H             ; Non-specific EOI command
     OUT  20H, AL             ;  to 8259A
     POP  DS
     POP  DI
     POP  DX
     POP  BX
     POP  AX
     IRET
_SERIAL_IN ENDP

_CHK_N_DISPLAY PROC NEAR
     PUSH DI
     IN   AL, 21H
     OR   AL, 10H             ; Disable 8259A IR4 in critical region
     OUT  21H, AL             ;  by masking bit 4 of IMR
     MOV  DI, HEAD_POINTER
     CMP  DI, TAIL_POINTER    ; Is queue empty ?
     JE   NOCHAR              ; Yes, just return
     MOV  AL, QUEUE[DI]       ; No, get char from queue to AL
     INC  DI                  ; Point DI at next byte in queue
     CMP  DI, 1000            ; See if time to wrap pointer around
     JNE  OK                  ; No, go on
     MOV  DI, 0               ; Yes, wrap pointer around to start
OK:
     MOV  HEAD_POINTER, DI    ; Store new pointer value
     PUSH AX                  ; Save character in AL on stack
     IN   AL, 21H
     AND  AL, 0ECH            ; Enable IR4 interrupt so new char in 8250
     OUT  21H, AL             ;  can interrupt INT 10H
     POP  AX                  ; Get character back from stack
     MOV  AH, 14              ; Use BIOS INT 10H to send to CRT
     MOV  BH, 0
     INT  10H
     DEC  _NUMCHAR            ; Dec number char sent to CRT
     JMP  DONE
NOCHAR:
     IN   AL, 21H
     AND  AL, 0ECH            ; End of critical region. Enable IR4 by
                              ;  unmasking bit 4 in IMR of 8259A so
     OUT  21H, AL             ;  new char in UART can interrupt
DONE:
     POP DI
     RET
_CHK_N_DISPLAY ENDP

_TEXT     ENDS
     END
```

FIGURE 14-27 *(Continued)*

The processor multiplies the type number by 4 and goes to that address in the interrupt-vector table to get the starting address of the service procedure for that interrupt. In your program you must in some way put the starting address of your interrupt-service procedure in the correct address in the interrupt-vector table.

The setvect(12, serial_in); statement near the start of our program calls a predefined function to initialize the int 12 location in the vector table with the starting address of our SERIAL_IN procedure. In a simple application such as this, the setvect statement is all that is needed, but we used the opportunity to show you an important technique that is used in "terminate and stay resident" programs, which we discuss more in the next chapter.

The point here is that whenever you write a program which changes some basic system parameter such as the contents of the interrupt-vector table, you should save the initial values of the parameters so you can restore them when your program finishes executing. There are three steps in saving and restoring an interrupt vector. The first step is to declare a pointer to an interrupt function with a statement such as void interrupt (oldcomrxint) () ;. In this statement interrupt is a special pointer type and oldcomrxint is the name we gave to the pointer. The second step in the process is to save the initial contents of the interrupt vector table with the oldcomrxint-getvect(12); statement. In this statement the predefined function getvect () copies the interrupt vector to the location pointed to by oldcomrxint. The final step is to restore the initial vector when our program terminates. We do this with the setvect (12,oldcomrxint); statement in our shutdown function. As we will show you later, the point of all this is that you can "intercept" a system interrupt such as the keyboard interrupt, use it for your own program, and then restore the system interrupt vector when your program finishes.

To do the rest of the initialization for this program we call the assembly language program INIT, so take a look at it now. The 8259A itself is mostly initialized by the BIOS when the system is turned on. However, since the UART is connected to IR4 of the 8259A, that input has to be unmasked. To do this the current contents of the 8259A interrupt mask register are read in from address 21H. The bit corresponding to IR4 is then ANDed with a 0 to unmask the interrupt, and the result is sent back to the interrupt mask register. Using this approach saves the system environment. It is important to do this rather than just sending out a control word directly, so that you don't disable other system functions. In this system, for example, the system clock tick is connected to IR0 and the keyboard is connected to IR1, so these would be disabled if you accidentally put 1's in these bits of the control word.

Initializing the 8250 UART is next. Figure 14-28, page 516, shows the internal addresses and the bit formats for the control words we need here. The first part of the initialization involves the baud rate, parity, and stop bits. Since this step requires several words to be sent, we simply used the BIOS INT 14H procedure to do it.

> NOTE: We initialize the 8250 here for 4800 Bd, so the baud rate jumper on the SDK-86 must be set for this baud rate.

The next task we do here is enable the desired interrupt circuitry in the 8250. In order to do this, the DLAB bit of the line control word must first be made a 0. Note that this is done by reading in the line control word, resetting the desired bit, and sending the word out again. This preserves the previous state of the rest of the bits in the line control register. As shown in Figure 14-28*b*, with DLAB = 0, a control word which enables the enable-receive line status interrupt can be sent to the interrupt enable register at address 03F9H. As shown in Figure 14-28*b*, the 8250 has four different conditions which can be enabled to assert the interrupt output when true. In cases where multiple interrupts are used, the interrupt identification register can be read to determine the source of an interrupt. For this application we are using only the enable receive line status interrupt, so a 1 is put in that bit. The final step in the 8250 initialization is to assert the $\overline{RTS}$, $\overline{DTR}$, and $\overline{OUT2}$ output signals. As shown by the circuit connections in Figure 14-10*b*, asserting $\overline{RTS}$ is necessary to assert the $\overline{CTS}$ input so the UART can transmit. Likewise, asserting $\overline{DTR}$ is necessary to assert the $\overline{DSR}$ and $\overline{CD}$ inputs of the UART. The $\overline{OUT2}$ signal from the 8250 must be asserted in order to enable a three-state buffer which is in series with the interrupt signal from the 8250 to the 8259A.

When you are working out an initialization sequence such as this, read the data sheet carefully and check out the actual hardware circuitry for the system you are working on. We missed the $\overline{OUT2}$ connection the first time through, but a second look at the schematic for the communications board showed that it was necessary to assert this signal. Now let's see how the procedure which reads characters from the UART works.

THE SERIAL_IN PROCEDURE

The purpose of the SERIAL_IN interrupt procedure is to read characters in from the UART and put them in a buffer. Note that since this is an interrupt procedure which can occur at any time, it is important to save the DS register of the interrupted program and load the DS register with DGROUP, the value needed for this procedure.

The buffer used here is a special type of queue called a *circular buffer* or *ring buffer*. Figure 14-29, page 517, attempts to show how this works. One pointer, called the TAIL_POINTER, is used to keep track of where the next byte is to be written to the buffer. Another pointer called the HEAD_POINTER is used to keep track of where the next character to be read from the buffer is located. The buffer is circular because when the tail pointer reaches the highest location in the memory space set aside for the buffer, it is "wrapped around" to the beginning of the buffer again. The head pointer follows the tail pointer around the circle as characters are read from the buffer. Two checks are made on the tail pointer before a character is written to the buffer.

First the tail pointer is brought into a register and incremented. This incremented value is then compared with the maximum number of bytes the buffer can hold. If the values are equal, the pointer is at the highest address in the buffer, so the register is reset to zero. After the current character is put in the buffer, this value will be loaded into TAIL_POINTER to wrap around to the lowest address in the buffer again.

Second, a check is made to see if the incremented value of the tail pointer is equal to the head pointer. If the two are equal, it means that the current byte can be written, but that the next byte would be written over the byte at the head of the queue. In this case we simply

I/O DECODE (IN HEX)		REGISTER SELECTED	DLAB STATE
PRIMARY ADAPTER	ALTERNATE ADAPTER		
3F8	2F8	TX BUFFER	DLAB=0 (WRITE)
3F8	2F8	RX BUFFER	DLAB=0 (READ)
3F8	2F8	DIVISOR LATCH LSB	DLAB=1
3F9	2F9	DIVISOR LATCH MSB	DLAB=1
3F9	2F9	INTERRUPT ENABLE REGISTER	DLAB=X
3FA	2FA	INTERRUPT IDENTIFICATION REGISTERS	DLAB=X
3FB	2FB	LINE CONTROL REGISTER	DLAB=X
3FC	2FC	MODEM CONTROL REGISTER	DLAB=X
3FD	2FD	LINE STATUS REGISTER	DLAB=X
3FE	2FE	MODEM STATUS REGISTER	DLAB=X

(a)

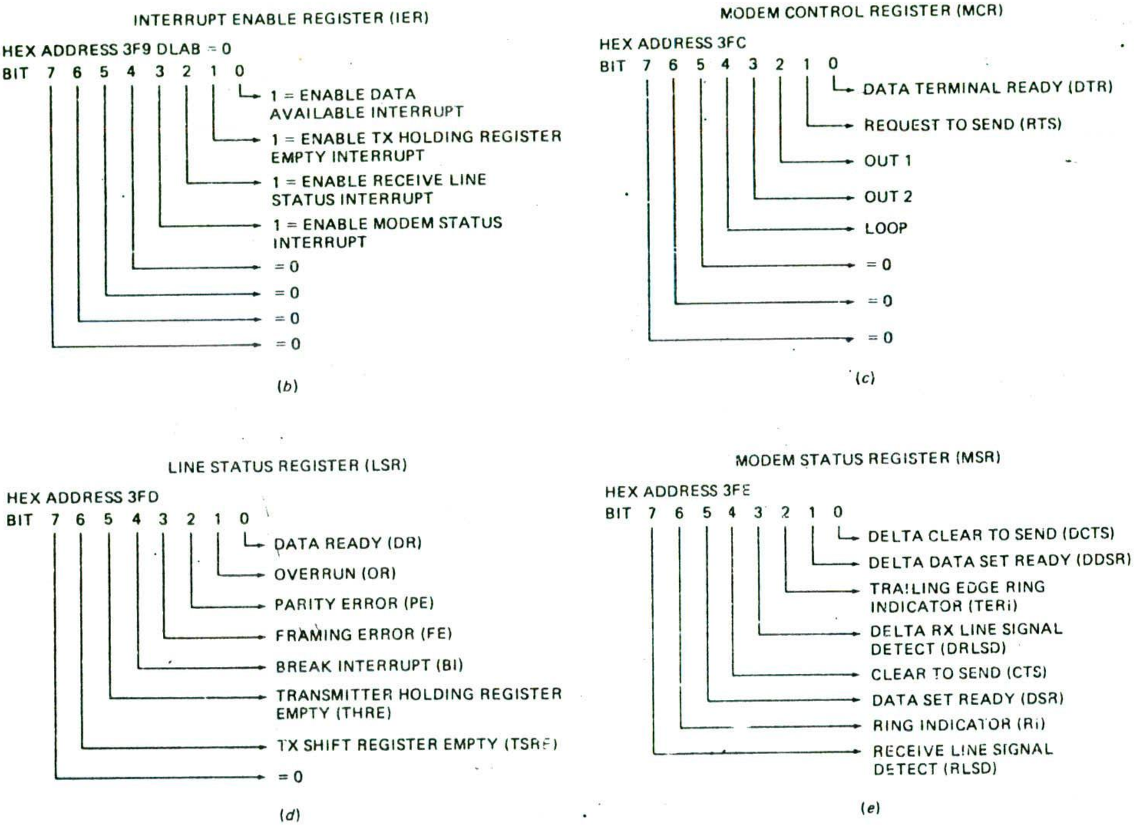

INTERRUPT ENABLE REGISTER (IER)
HEX ADDRESS 3F9 DLAB = 0
BIT 7 6 5 4 3 2 1 0
1 = ENABLE DATA AVAILABLE INTERRUPT
1 = ENABLE TX HOLDING REGISTER EMPTY INTERRUPT
1 = ENABLE RECEIVE LINE STATUS INTERRUPT
1 = ENABLE MODEM STATUS INTERRUPT
= 0
= 0
= 0
= 0
(b)
MODEM CONTROL REGISTER (MCR)
HEX ADDRESS 3FC
BIT 7 6 5 4 3 2 1 0
DATA TERMINAL READY (DTR)
REQUEST TO SEND (RTS)
OUT 1
OUT 2
LOOP
= 0
= 0
= 0
(c)
LINE STATUS REGISTER (LSR)
HEX ADDRESS 3FD
BIT 7 6 5 4 3 2 1 0
DATA READY (DR)
OVERRUN (OR)
PARITY ERROR (PE)
FRAMING ERROR (FE)
BREAK INTERRUPT (BI)
TRANSMITTER HOLDING REGISTER EMPTY (THRE)
TX SHIFT REGISTER EMPTY (TSRE)
= 0
(d)
MODEM STATUS REGISTER (MSR)
HEX ADDRESS 3FE
BIT 7 6 5 4 3 2 1 0
DELTA CLEAR TO SEND (DCTS)
DELTA DATA SET READY (DDSR)
TRAILING EDGE RING INDICATOR (TERI)
DELTA RX LINE SIGNAL DETECT (DRLSD)
CLEAR TO SEND (CTS)
DATA SET READY (DSR)
RING INDICATOR (RI)
RECEIVE LINE SIGNAL DETECT (RLSD)
(e)

FIGURE 14-28 8250 addresses in IBM PC, registers, and control words. (*a*) System addresses. (*b*) Interrupt enable register. (*c*) Modem control register. (*d*) Line status register. (*e*) Modem status register.

return to the interrupted program without writing the current character into the buffer. Actually this wastes a byte of buffer space, but it is necessary to do this so that the pointers have different values for this buffer-full condition than they do for the buffer-empty condition. The buffer-empty condition is indicated when the head pointer is equal to the tail pointer. If the buffer is not full, the character read in from the UART is written to the buffer, and the pointer to the next available location in the buffer is transferred from the register to the memory location called TAIL_POINTER. Finally, before returning, an end-of-interrupt command must be sent to the 8259A to reset bit 4 of the interrupt service register.

To summarize the operation of a circular buffer, then, bytes are put in at the tail pointer location and read out

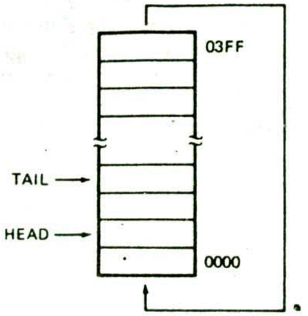


FIGURE 14-29 Diagram showing how ring buffer pointers wrap around at the top of the allocated buffer space.

from the head pointer location. The buffer is considered full when the tail pointer reaches one less than the head pointer. The buffer is empty when the head pointer is equal to tail pointer.

THE CHK_N_DISPLAY PROCEDURE

The main purpose of the CHK_N_DISPLAY procedure is to read a character from the circular buffer and send it to the CRT with the BIOS INT 10H procedure. In order to make sure the procedure operates correctly under all conditions, however, we mask the IR4 interrupt in the 8259A right at the start so that an interrupt from the UART cannot call the SERIAL_IN procedure while CHK_N_DISPLAY is using the head and tail pointers. This is necessary to prevent the SERIAL_IN procedure from altering the values of the pointers in the middle of CHK_N_DISPLAY's use of them and causing the CHK_N_DISPLAY procedure to make the wrong decisions about whether the buffer is empty, for example. The group of instructions which you need to protect from interruption is called a *critical region*. It is important to keep critical regions as short as possible so that interrupts need not be masked for unnecessarily long times. Note that we masked the IR4 interrupt input of the 8259A rather than disable the processor interrupt. This was done so that the keyboard and the timer interrupts, which have nothing to do with the critical region in this procedure, can keep running.

Once the critical region is safe, a check is made to see if there are any characters in the buffer. If not, the 8259A IR4 input is unmasked, and execution returned to the calling program. If a character is available in the buffer, the character is read out and the head pointer updated to point to the next available character. If the pointer is at the top of space allotted for the buffer, the pointer will be wrapped around to the start of the buffer again. As soon as the character is read out from the buffer and the pointers updated, an interrupt from the UART cannot do any damage, so we unmask IR4. The BIOS INT 10H procedure is then used to send the character to the CRT. If a UART interrupt occurs during the INT 10H procedure, the SERIAL_IN procedure will read the character from the UART and return execution to the INT 10H procedure. This short interruption produces no noticeable effect on the operation of the INT 10H procedure, and it makes sure no characters from the UART are missed. After the INT 10H procedure finishes, a character-sent counter called NUMCHAR is decremented and execution is returned to the calling program. This counter counts the number of characters actually sent to the CRT rather than just the number of times the CHK_N_DISPLAY procedure is called. This allows the procedure to be called over and over again until a given number of characters are received from the SDK-86 and sent to the CRT.

THE XMIT FUNCTION

After first checking to see if the UART transmitter buffer is ready, the XMIT procedure sends a character to the 8250 UART. To determine if the UART transmitter buffer is empty, we use the predefined bioscom () function, which is just a simple way to call the BIOS INT 14H procedure. The first argument in the bioscom parentheses is the BIOS INT 14H subfunction number. The second argument is the value you want to pass to the BIOS INT 14H procedure in AL. The third argument in the parentheses is the COM port number, 0 for COM1 and 1 for COM2. As you can see in Figure 14-24, if you call the BIOS INT14H procedure with AH = 3, the status of the UART will be returned in AX. Our while loop repeats until the transmitter holding register bit, bit 13, becomes a 1. Normally, you should also check that CTS in bit 4 and DSR in bit 5 are also asserted. For this program, however, these signals are asserted by jumpers on the connector, so we didn't bother to check them.

To send the character to the UART we did a direct write to the UART transmitter holding register with the outportb(0x3fc, ascval) statement. Normally we avoid direct writes such as this and use BIOS procedures, DOS function calls, or C function calls so the program is more likely to run on a variety of systems. However, when we tried to use the statement bioscom(1,ascval,0) to send a character to the UART, the program would no longer read characters sent to the UART by the SDK-86. A careful reading of the BIOS INT 14H procedure in the IBM Technical Reference Manual showed that the Transmit subprocedure of the BIOS INT 14H procedure turns off the OUT2 signal. As we described in the initialization section, this signal must be asserted so that receiver-ready interrupt signals can get from the UART to the 8259A and call the SERIAL-IN procedure. The only real cure we found for the problem was to do the direct write to the UART as shown. Incidentally, the bioscom() function works fine if you are both sending and receiving characters on a polled basis.

THE CONVERT_AND_SEND FUNCTION

The SDK-86 requires that each nibble of a program code byte be sent in as the corresponding ASCII character. The code byte 3AH, for example, must be sent as 33H (ASCII 3), followed by 41H (ASCII A). We converted this procedure to C to show you how you can do bit operations in C. You can work your way through this section with an example data byte to see how it works. After the ASCII characters for each code byte are sent, the ASCII code for a comma is sent, as required by the SDK-86, and execution is returned to the L section of the switch structure. There the SDK-86 response is sent to the CRT.

THE SHUTDOWN FUNCTION

As we said earlier, you should always put everything back in its initial state when your program finishes executing. The shutdown function here remasks the IR4 input of the 8259A by reading the current value, ORing that value with 10H to set bit 4, and sending the result back to the 8259A. The setvect(12,oldcomrxint) function call restores the initial interrupt vector to the INT 12 location in the vector table. Finally, the exit (0) function call closes any open files and returns execution to DOS.

CONCLUSION

This program was written to do a specific job and to demonstrate such important programming concepts as installing an interrupt vector in a C program, interacting with a UART, working with a ring buffer, reading binary files, and preserving the system environment. Space limitations prevented us from making the program as "friendly" as we would have liked it to be. Perhaps you can see how the program could easily be modified to, for example, let the user enter the desired communications port number and the desired baud rate.

SYNCHRONOUS SERIAL-DATA COMMUNICATION AND PROTOCOLS

Introduction

Most of the discussion of serial-data transfer up to this point in the chapter has been about asynchronous transmission. For asynchronous serial transmission, a start bit is used to identify the beginning of each data character, and at least one stop bit is used to identify the end of each data character. The transmitter and the receiver are effectively synchronized on a character-by-character basis. With a start bit, 1 stop bit, and 1 parity bit, a total of 10 bits must be sent for each 7-bit ASCII character. This means that 30 percent of the transmission time is wasted. A more efficient method of transferring serial data is to synchronize the transmitter and the receiver and then send a large block of data characters one after the other with no time between characters. No start or stop bits are then needed with individual data characters, because the receiver automatically knows that every 8 bits received after synchronization represents a data character. When a block of data is not being sent through a synchronous data link, the line is held in a marking condition. To indicate the start of a transmission, the transmitter sends out one or more unique characters called *sync characters* or a unique bit pattern called a *flag*, depending on the system being used. The receiver uses the sync characters or the flag to synchronize its internal clock with that of the receiver. The receiver then shifts in the data following the sync characters and converts them to parallel form so they can be read in by a computer. As we said in the discussions of modems and ISDN, high-speed modems and digital communication channels use synchronous transmission.

Now, remember from a previous section that a hardware level set of handshake signals is required to transmit asynchronous or synchronous digital data over phone lines with modems. In addition to this handshaking, a higher level of coordination, or *protocol*, is required between transmitter and receiver to assure the orderly transfer of data. A protocol in this case is an agreed set of rules concerning the form in which the data is to be sent. There are many different serial data protocols. The two most common that we discuss here are the IBM *binary synchronous communications protocol*, or BISYNC, and the *high-level data link control protocol*, or HDLC.

Binary Synchronous Communication Protocol—BISYNC

BISYNC is referred to as a *byte-controlled protocol* (BCP), because specified ASCII or EBCDIC characters (bytes) are used to indicate the start of a message and to handshake between the transmitter and the receiver. Incidentally, even in a full-duplex system, BISYNC protocol only allows data transfer in one direction at a time.

Figure 14-30 shows the general message format for BISYNC. For our first cycle through this we will assume that the transmitter has received a message from the receiver that it is ready to receive a transmission. If no message is being sent, the line is an "idle" condition with a continuous high on the line. To indicate the start of a message, the transmitting system sends two or more previously agreed upon sync characters. For example, a sync character might be the ASCII 16H. As we said before, the receiver uses these sync characters to synchronize its clock with that of the transmitter. A header may then be sent if desired. The header contents are usually defined for a specific system and may include information about the type, priority, and destination of the message that follows. The start of the header is indicated with a special character called *start-of-header* (SOH), which in ASCII is represented by 01H.

After the header, if present, the beginning of the text portion of the message is indicated by another special character called *start-of-text* (STX), which in ASCII is represented by 02H. To indicate the end of the text portion of the message, an *end-of-text* (ETX) character or an *end-of-block* (ETB) character is sent. The text portion may contain 128 or 256 characters (different systems use different-size blocks of text). Immediately following the ETX, character 1 or 2 block check charac-

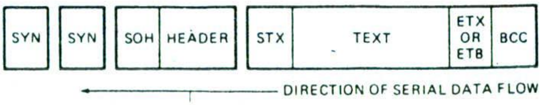


FIGURE 14-30 General message format for binary synchronous communication (BISYNC).

ters (BCC) are sent. For systems using ASCII, the BCC is a single byte which represents complex parity information computed for the text of the message. For systems using EBCDIC, a 16-bit *cyclic redundancy check* is performed on the text part of the message and the 16-bit result sent as 2 BCCs. The point of these BCCs is that the receiving system can recompute the value for them from the received data and compare the results with the BCCs sent from the transmitter. If the BCCs are not equal, the receiver can send a message to the transmitter, telling it to send the message again. Now let's look at how messages are used for data transfer handshaking between the transmitter and the receiver.

To start let's assume that we have a remote "smart" terminal connected to a computer with a half-duplex connection. Further, let's assume that the computer is in the receive mode. Now, when the program in the terminal determines that it has a block of data to send to the computer, it first sends a message with the text containing only the single character ENQ (ASCII 05H), which stands for *enquiry*. The terminal then switches to receive mode to await the reply from the computer. The computer reads the ENQ message, and, if it is not ready to receive data, it sends back a text message containing the single character for *negative acknowledge*, NAK (ASCII 15H). If the receiver is ready, it sends a message containing the *affirmative acknowledge*, ACK, character (ASCII 06H). In either case, the computer then switches to receive mode to await the next message from the terminal. If the terminal received a NAK, it may give up, or it may wait a while and try again. If the terminal received an ACK, it will send a message containing a block of text and ending with a BCC character(s). After sending the message, the terminal switches to receive mode and awaits a reply from the computer as to whether the message was received correctly. The computer meanwhile computes the BCC for the received block of data and compares it with the BCC sent with the message. If the two BCCs are not equal, the computer sends a NAK message to the terminal. This tells the terminal to send the message again, because it was not received correctly. If the two BCCs are equal, then the computer sends an ACK message to the terminal, which tells it to send the next message or block of text. In a system where multiple blocks of data are being transferred, an ACK 0 message is usually sent for one block, an ACK 1 message sent for the next, and an ACK 0 again sent for the next. The alternating ACK messages are a further help in error checking. In either case, after the message is sent the computer switches to receive mode to await a response from the terminal.

A variation of BISYNC commonly used to transfer binary files in the PC environment and between Unix systems and PCs is called *XMODEM*. An XMODEM block consists of a SOH character, a block number, 128 bytes of data (padded if necessary to fill the block), and an 8-bit checksum. A transmission starts with the receiver sending a NAK character to the sender. The sender then sends a block of data. If the data is received correctly, the receiver sends back an ACK and the sender sends the next block of data. If the data is not received correctly, the receiver sends a NAK and the sender sends the block again. The transmission is completed when the sender sends an end-of-transmission (EOT) character and the receiver replies with an ACK.

One major problem with a BISYNC type protocol is that the transmitter must stop after each block of data is transferred and wait for an ACK or NAK signal from the receiver. Due to the wait and line turnaround times, the actual data transfer rate may be only half of the theoretical rate predicted by the physical bit rate of the data link. The HDLC protocol discussed in a later section greatly reduces this problem. Next we want to return to the Intel 8251A USART which is used on the IBM PC Synchronous Communication Adapter and give you a brief look at how it is used for BISYNC communication.

USING THE INTEL 8251A USART FOR BISYNC COMMUNICATION

As shown in Figure 14-5, we initialize an 8251A by first getting its attention, sending it a mode word, and then sending it a command word. To initialize the 8251A for synchronous communication, 0's are put in the least significant 2 bits of the mode word. The rest of the bits in the mode word then have the meanings shown in Figure 14-31*a*, page 520. Most of the bit functions should be reasonably clear from the descriptions in the figure, but a couple need a little more explanation.

Bit 6 of the mode word specifies the SYNDET pin on the 8251A to be an input or an output. The pin is programmed to function as an input if external circuitry is used to detect the sync character in the data bit stream. When programmed as an output, the pin will go high when the 8251A has found one or more sync characters in the data bit stream.

Bit 7 of the mode word is used to specify whether 1 sync character or a sequence of 2 different sync characters is to be looked for at the start of a message.

To initialize an 8251A for synchronous operation:

1. Send a series of nulls and a software reset command to the control address as shown at the start of Figure 14-5.
2. Send a mode word based on the format in Figure 14-31*a* to the control address.
3. Send the desired sync character for that particular system to the control address of the 8251A.
4. If a second sync character is needed, send it to the control address.
5. Finally, send a command word to the control address to enable the transmitter, enable the receiver, and enable the device to look for sync characters in the data bit stream coming in the RxD input.

The format for the command word is shown in Figure 14-31*b*. Now, let's examine how the 8251A participates in a synchronous data transfer. As you work your way through this section, try to keep separate in your mind the parts of the process that are done by the 8251A and the parts that are done by software at one end of the link or the other.

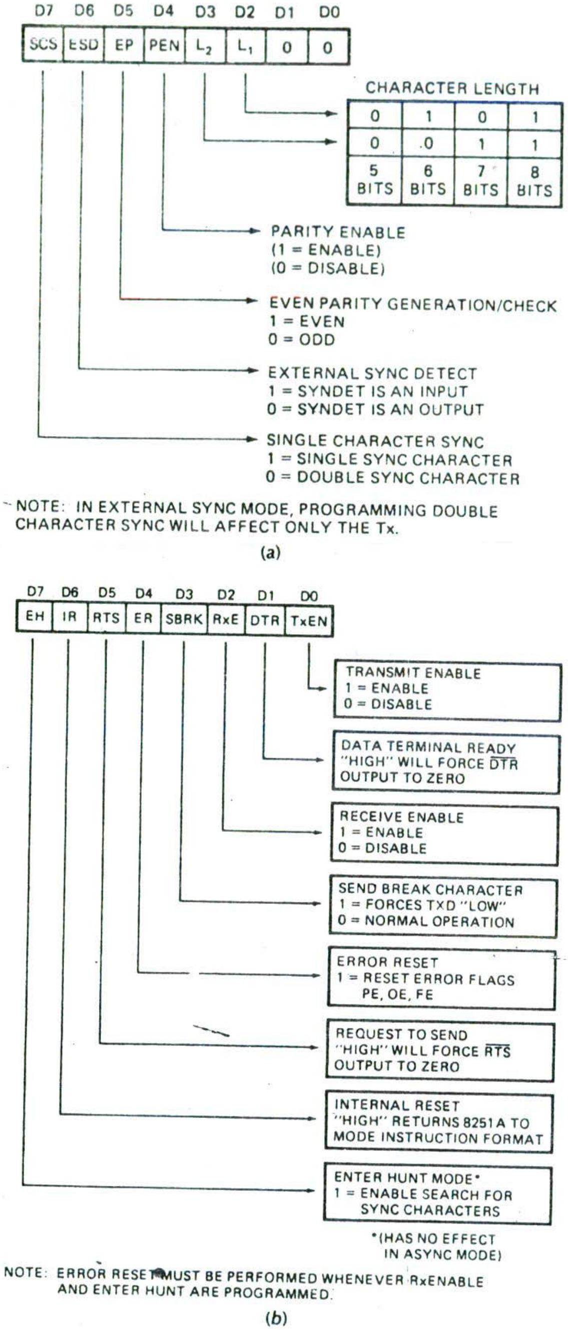

D7 D6 D5 D4 D3 D2 D1 D0
SCS ESD EP PEN L2 L1 0 0
CHARACTER LENGTH
0 1 0 1
0 0 1 1
5 BITS 6 BITS 7 BITS 8 BITS
PARITY ENABLE
(1 = ENABLE)
(0 = DISABLE)
EVEN PARITY GENERATION/CHECK
1 = EVEN
0 = ODD
EXTERNAL SYNC DETECT
1 = SYNDET IS AN INPUT
0 = SYNDET IS AN OUTPUT
SINGLE CHARACTER SYNC
1 = SINGLE SYNC CHARACTER
0 = DOUBLE SYNC CHARACTER
NOTE: IN EXTERNAL SYNC MODE, PROGRAMMING DOUBLE CHARACTER SYNC WILL AFFECT ONLY THE Tx.
(a)
D7 D6 D5 D4 D3 D2 D1 D0
EH IR RTS ER SBRK RxE DTR TxEN
TRANSMIT ENABLE
1 = ENABLE
0 = DISABLE
DATA TERMINAL READY
"HIGH" WILL FORCE DTR OUTPUT TO ZERO
RECEIVE ENABLE
1 = ENABLE
0 = DISABLE
SEND BREAK CHARACTER
1 = FORCES TXD "LOW"
0 = NORMAL OPERATION
ERROR RESET
1 = RESET ERROR FLAGS PE, OE, FE
REQUEST TO SEND
"HIGH" WILL FORCE RTS OUTPUT TO ZERO
INTERNAL RESET
"HIGH" RETURNS 8251A TO MODE INSTRUCTION FORMAT
ENTER HUNT MODE*
1 = ENABLE SEARCH FOR SYNC CHARACTERS
*(HAS NO EFFECT IN ASYNC MODE)
NOTE: ERROR RESET MUST BE PERFORMED WHENEVER RxENABLE AND ENTER HUNT ARE PROGRAMMED.
(b)

FIGURE 14-31 8251A synchronous mode and command word formats. (*a*) Mode word. (*b*) Command word. (Intel)

To start, let's assume the 8251A is in a terminal which has blocks of data to send to a computer, as we described earlier in this section. Further assume that the computer is in receive mode waiting for a transmission from the terminal and that the 8251A in the terminal has been initialized and is sending out a continuous high on the TxD line.

An I/O driver routine in the terminal will start the transfer process by sending a sync character(s), SOH character, header characters, STX character, ENQ character, ETX character, and BCC byte to the 8251A, one after the other. The 8251A sends the characters out in synchronous serial format (no start and stop bits). If, for some reason such as a high-priority interrupt, the CPU stops sending characters while a message is being sent, the 8251A will automatically insert sync characters until the flow of data characters from the CPU resumes.

After the ENQ message has been sent, the CPU in the terminal awaits a reply from the computer through the RxD input of the 8251A. If the 8251A has been programmed to enter hunt mode by sending it a control word with a 1 in bit 7, it will continuously shift in bits from the RxD line and check after each shift if the character in the receive buffer is a sync character. When it finds a sync character, the 8251A asserts the SYNDET pin high, exits the hunt mode, and starts the normal data read operation. When the 8251A has a valid data character in its receiver buffer, the RxRDY pin will be asserted, and the RxRDY bit in the status register will be set. Characters can then be read in by the CPU on a polled or an interrupt basis.

When the CPU has read in the entire message, it can determine whether the message was a NAK or an ACK. If the message was an ACK, the CPU can then send the actual data message sequence of characters to the 8251A. Handshake and data messages will be sent back and forth until all the desired block of data has been sent to the computer. In the next section we discuss another protocol used for synchronous serial-data transfer.

High-level Data Link Control (HDLC) and Synchronous Data Link Control (SDLC) Protocols

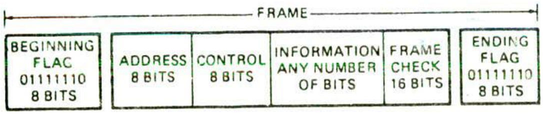
The BISYNC-type protocols which we discussed in the previous section work only in half-duplex mode; except for XMODEM, they have difficulty transmitting pure 8-bit binary data such as object code for programs; and they are not easily adapted to serving multiple units sharing a common data link. In an attempt to solve these problems, the *International Standards Organization* (ISO) proposed the *high-level data link control protocol* (HDLC) and IBM developed the *synchronous data link control protocol* (SDLC). The standards are so nearly identical that, for the discussion here, we will treat them together under the name HDLC and indicate any significant differences as needed.

As we said previously, BISYNC is referred to as a byte-controlled protocol because character codes or bytes such as SOH, STX, and ETX are used to mark off parts of a transmitted message or act as control messages. HDLC is referred to as a *bit-oriented protocol* (BOP) because messages are treated simply as a string of bits rather than a string of characters. The group of bits which make up a message is referred to as a *frame*. The three types of frames used are information or *I frames*, supervisory control sequences or *S frames*, and command/response or *U frames*. The three types of frames all have the same basic format.

Figure 14-32*a* shows the format of an HDLC frame. Each part of the frame is referred to as *field*. A frame starts and ends with a specific bit pattern, 01111110, called a *flag* or *flag field*. When no data is being sent, the line idles with all 1's, or continuous flags. Immediately after the flag field is an 8-bit address field which contains the address of the destination unit for a control or information frame and the source of the response for a response frame.

Figure 14-32*b* shows the meaning of the bits in the 8-bit control field for each of the three types of frames. We don't have the space or the desire to explain here the meaning of all of these. A little later we will, however, explain the use of the Ns and Nr bits in the control byte for an information frame.

The information field, which is present only in information frames, can have any number of bits in HDLC protocol, but in SDLC the number of bits has to be a multiple of 8. In some systems as many as 10,000 or 20,000 information bits may be sent per frame. Now, the question may occur to you, What happens if the data contains the flag bit pattern, 01111110? The answer to this question is that a special hardware circuit modifies the bit stream between flags so that there are never more than five 1's in sequence. To do this the circuit monitors the data stream and automatically stuffs in a 0 after any series of five 1's. A complementary circuit in the receiver removes the extra zeros. This scheme allows character codes or binary data to be sent without the problems BISYNC has in this area.


FRAME
BEGINNING FLAC 01111110 8 BITS
ADDRESS 8 BITS
CONTROL 8 BITS
INFORMATION ANY NUMBER OF BITS
FRAME CHECK 16 BITS
ENDING FLAG 01111110 8 BITS

(*a*)

BITS IN CONTROL FIELD

HDLC FRAME FORMAT	7	6	5	4	3	2	1	Ø
I-FRAME (INFORMATION TRANSFER COMMANDS/RESPONSES)	Nr	Nr	Nr	P/F	Ns	Ns	Ns	Ø
S-FRAME (SUPERVISORY COMMANDS/RESPONSES)	Nr	Nr	Nr	P/F	S	S	Ø	1
U-FRAME (UNNUMBERED COMMANDS/RESPONSES)	M	M	M	P/F	M	M	1	1

SENDING ORDER - BIT Ø FIRST, BIT 7 LAST

NS THE TRANSMITTING STATION SEND SEQUENCE NUMBER, BIT 2 IS THE LOW-ORDER BIT

P/F THE POLL BIT FOR PRIMARY STATION TRANSMISSIONS, AND THE FINAL BIT FOR SECONDARY STATION TRANSMISSIONS

Nr THE TRANSMITTING STATION RECEIVE SEQUENCE NUMBER, BIT 6 IS THE LOW-ORDER BIT

S THE SUPERVISORY FUNCTION BITS

M THE MODIFIER FUNCTION BITS

(*b*)

FIGURE 14-32 (*a*) Format of HDLC frame. (*b*) Meaning of bits in 8-bit control field of frame.

The next field in a frame is the 16-bit *frame check sequence* (FCS). This is a cyclic redundancy word derived from all the bits between the beginning and end flags, but not including 0's inserted to prevent false flag bytes. This CRC value is recomputed by the receiving system to check for errors.

Finally, a frame is terminated by another flag byte. The ending flag for one frame may be the starting flag for another frame.

In order to describe the HDLC data transfer process, we first need to define a couple of terms. HDLC is used for communication between two or more systems on a data link. One of the systems or *stations* on the link will always be set up as a controller for the link. This station is called the *primary station*. Other stations on the link are referred to as *secondary stations*.

Now, suppose that a primary station—a computer, for example—wants to send several frames of information to a secondary station such as another computer or terminal. Here's how a transfer might take place.

The primary station starts by sending an S frame containing the address of the desired secondary station and a control word which inquires if the receiver is ready. The secondary station then sends an S frame which contains the address of the primary station and a control word which indicates its ready status. If the secondary station receiver was ready, the primary station then sends a sequence of information frames. The information frames contain the address of the secondary station, a control word, a block of information, and the FCS words. For all but the last frame of a sequence of information frames, the P/F bit in the control byte will be a 0. The 3 Ns bits in the control byte will contain the number of the frame in the sequence.

Now, as the secondary station receives each information frame, it reads the data into memory and computes the frame-check sequence for the frame. For each frame in a sequence that the secondary station receives correctly, it increments an internal counter. When the primary station sends the last frame in a sequence of up to seven frames, it makes the P/F bit in the control byte a 1. This is a signal to the secondary station that the primary station wants a response as to how many frames were received correctly. The secondary station responds with an S frame. The Nr bits in the control word of this S frame contain the sequence number of the last frame that was received correctly plus 1. In other words, Ns represents the number of the next expected frame. The primary station compares Ns = 1 with the number of frames sent in the sequence. If the two numbers do not agree, the primary station knows that it must retransmit some frames, because they were not all received correctly. The Nr number tells the primary station which frame number to start the retransmission from. For example, if Nr is 3, the primary station will retransmit the sequence of frames starting with frame 3. If the sequence of frames was received correctly, another series of frames can be sent if desired. Actually, since HDLC operates in full-duplex, the receiving station can be queried after each frame is sent to see if the previous frame was received correctly. A similar series of actions takes place when a secondary station transmits to a primary station or to another secondary station.

One advantage of this HDLC scheme is that a large number of bits can be sent in a frame so the framing bit percentage is low. Another advantage is that the transmitter does not have to stop after every short message for an acknowledge as it does in BISYNC protocol. True, several frames may have to sent again in case of an error, but in low-error-rate systems, this is the exception. HDLC is often used with high-speed modems, and as we will show in the next section, HDLC is used along with some higher-level protocols for network communication between a wide variety of systems.

A final point to discuss here is how HDLC protocol is implemented with a microcomputer. At the basic hardware level, a standard USART cannot be used because of the need to stuff and strip 0 bits. Instead, specially designed parts such as the Intel 8273 HDLC/SDLC protocol controller are used. Devices such as this automatically stuff and strip the required 0 bits, generate and check frame-check sequence words, and produce the interface signals for RS-232C. The devices interface directly to microcomputer buses.

The actual control of which station uses the data link at a particular time and the formatting of frames is done by the system software. The next section discusses how several systems can be connected together, or "networked," so they can communicate with each other.

LOCAL AREA NETWORKS

Introduction

The objective of this section is to show you how several computers can be connected together to communicate with each other and to share common peripherals such as printers, large disk drives, FAX machines, etc. We will start with simple cases and progress to the type of network that might be used in the computerized electronics factory we described in an earlier chapter.

To communicate between a single terminal and a nearby computer, a simple RS-232C connection is sufficient. If the computer is distant, then a modem and phone line or a leased phone line is used, depending on the required data rate. Now, for a more difficult case, suppose that we have in a university building 100 terminals that need to communicate with a distant computer. We could use 100 phone lines with modems, but this seems quite inefficient. One solution to this problem is to run wires from all of the terminals to a central point in the building and then use a multiplexer or *data concentrator* of some type to send all the communications over one wideband line. Either time-domain multiplexing or frequency-division multiplexing can be used. A demultiplexer at the other end of the line reconstructs the original signals.

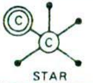
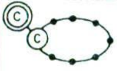
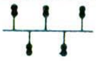
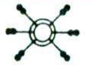
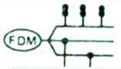
As another example of computer communication, suppose that we have several computers in one building or in a complex of buildings and that the computers need to communicate with each other. Our computerized electronics factory is an example of this situation. What is needed in this case is a high-speed network, commonly called a *local area network* or *LAN*, connecting the computers together. We start our discussion of LANs by showing you some of the basic ways that the systems on a network are connected together.

LAN Topologies

The different ways of physically connecting devices on a network with each other are commonly referred to as *topologies*. Figure 14-33 shows the five most common topologies and some other pertinent data about each, such as examples of commercially available systems which use each type.

In a *star topology* network, a central controller coordinates all communication between devices on the network. The most familiar example of how this works is probably a private automatic branch exchange, or PABX, phone system. In a PABX all calls from one phone on the system to another or to an outside phone are routed through a central switchboard. The new digital PABX systems allow direct communication between computers within a building at rates up to perhaps 100K bits/s.

In the *loop topology*, one device acts as a controller. If a device wants to communicate with one or more other devices on the loop, it sends a request to the controller.

TOPOLOGY	TYPICAL PROTOCOLS	TYPICAL NO. OF NODES	TYPICAL SYSTEMS
 STAR	RS-232C OR COMPUTER	TENS	PABX, COMPUTER- µC CLUSTERS STARLAN
 LOOP	SDLC	TENS	CPIB IBM 3600/3700, µC CLUSTERS
 COMMON BUS	CSMA/CD OR CSMA WITH ACKNOWLEDGMENT	TENS TO HUNDREDS PER SEGMENT	ETHERNET, 3COM OMNINET, Z-NET µC CLUSTERS
 RING	SDLC HDLC (TOKEN PASSING)	TENS TO HUNDREDS PER CHANNEL	PRIMENET, DOMAIN, OMNILINK µC CLUSTERS
 OTHER SERVICES BROADBAND BUS	CSMA/CD RS-232C & OTHERS PER CHANNEL	TWO TO HUNDREDS PER CHANNEL	· WANGNET, LOCALNET M/A-COM

● TERMINAL
⁞ DISTRIBUTED CONTROL
Ⓒ LOCAL CONTROLLER
◎ MULTINETWORK CONTROLLER
(FDM) FREQUENCY DIVISION MULTIPLEX

FIGURE 14-33 Summary of common computer network topologies.

If the loop is not in use, the controller enables the one device to output and the other device(s) to receive. The GPIB or IEEE 488 bus described in the last section of this chapter is an example of this topology.

In the *common-bus topology*, control of the bus is spread among all the devices on the bus. The connection in this type of system is simply a wire (usually but not always a coaxial cable), which any number of devices can be tapped into. Any device can take over the bus to transmit data. Data is transmitted in fixed-length blocks called *packets*. One common scheme to prevent two devices from transmitting at the same time is called *carrier sense, multiple access with collision detection*, or CSMA/CD. We discuss the details of CSMA/CD in a later section on Ethernet.

In a *ring network*, the control is also distributed among all the devices on the network. Each device on the ring functions as a repeater, which means that it simply takes in the data stream and passes the data stream on to the next device on the ring if it is not the intended receiver for the data. Data always circulates around the ring in one direction. Any device can transmit on the ring. A *token* is one common way used to prevent two or more devices from transmitting at the same time. A token is a specific lone byte such as 01111111 which is circulated around the ring when no device is transmitting. A device must possess the token in order to transmit. When a device needs to transmit, it removes the token from the bus, thus preventing any other devices from transmitting. After transmitting one or more packets of data, the transmitting device puts the token back on the ring so another device can grab it and transmit. We discuss this more in a later section.

The final topology we want to discuss here is the *tree* structured network, which often uses broadband transmission. Before we can really explain this one, we need to introduce you to a couple of terms commonly used with networks. In some networks such as Ethernet, data is transmitted directly as digital signals at rates of up to 10 Mbits/s. With this type of signal, only one device can transmit at a time. This form of data transmission is often referred to as *baseband* transmission, because only one basic frequency is used. The other common form of data transmission on a network is referred to as *broadband* transmission. Broadband transmission is based on a frequency-division multiplexing scheme such as that used for community antenna television (CATV) systems. The radio-frequency spectrum is divided up into 6-MHz-bandwidth channels.

A single device or group of devices can be assigned one channel for transmitting and another for receiving. Each channel or pair of channels is considered a branch on the tree. Special modems are used to convert digital signals to and from the modulated radio-frequency signals required. The multiple channels and the 6-MHz bandwidth of the channels in a broadband network allow voice, data, and video signals to be transmitted at the same time throughout the network. This is an advantage over baseband systems, which can transmit only one digital data signal at a time, but the broadband system is much more expensive.

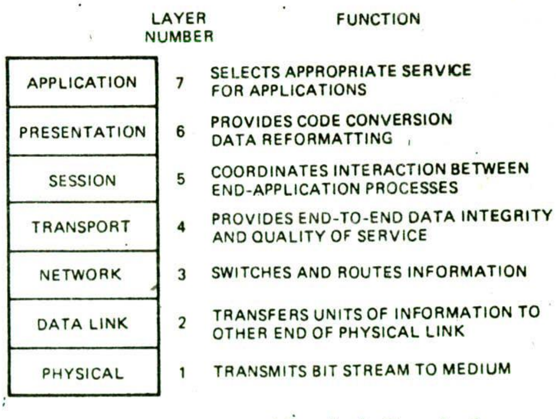
Network Protocols

In order for different systems on a network to communicate effectively with each other, a series of rules or protocols must be agreed upon and followed by all of the devices on the network. The International Standards Organization, in an attempt to bring some order to the chaos of network communication, has developed a set of standards called the *open systems interconnection* (OSI) model. This model is more of a recommendation than a rigid standard, but to increase compatibility more and more manufacturers are attempting to follow it. The OSI model is a seven-layer hierarchy of protocols as shown in Figure 14-34. This layered approach structures the design tasks and makes it possible to change, for example, the actual hardware used to transmit the data without changing the other layers. We will use a common network operation, electronic mail, to explain to you the function of the upper-layers model.

Electronic mail allows a user on one system on a network to send a message to another user on the same system or on another system. The message is actually sent to a "mailbox" in a hard-disk file. Each user on the network periodically checks a personal mailbox to see if it contains any messages. If any messages are present, they can be read out and then deleted from the mailbox.

The *application layer* of the OSI model specifies the general operation of network services such as electronic mail, file management, program-to-program communication, and peripheral sharing. For our electronic mail example, this layer of the protocol would specify the format for invoking the electronic mail function.

The *presentation layer* of the OSI protocol governs the programs which convert messages to the code and format that will be understood by the receiver. For our electronic mail message, this layer might involve translating the message from ASCII codes to EBCDIC codes [illegible] formatting the message into packets or frames such as those we described for HDLC in a previous



	LAYER NUMBER	FUNCTION
APPLICATION	7	SELECTS APPROPRIATE SERVICE FOR APPLICATIONS
PRESENTATION	6	PROVIDES CODE CONVERSION DATA REFORMATTING
SESSION	5	COORDINATES INTERACTION BETWEEN END-APPLICATION PROCESSES
TRANSPORT	4	PROVIDES END-TO-END DATA INTEGRITY AND QUALITY OF SERVICE
NETWORK	3	SWITCHES AND ROUTES INFORMATION
DATA LINK	2	TRANSFERS UNITS OF INFORMATION TO OTHER END OF PHYSICAL LINK
PHYSICAL	1	TRANSMITS BIT STREAM TO MEDIUM

FIGURE 14-34 International Standards Organization open systems interconnect (OSI) model for network communications.

section a standard file format. Data compression and encryption also fail in this layer of the protocol.

The *session layer* of the OSI protocol establishes and terminates logical connections on the network. This layer is responsible for opening and closing named files, for translating a user name into a physical network address, and checking passwords. Electronic mail allows you to specify the intended receiver of a message by name. It is the responsibility of this layer of the protocol to make the connection between the name and the network address of the named receiver.

The *transport layer* of the protocol is responsible for making sure a message is transmitted and received correctly. An example of the operation of this protocol layer is the ACK or NAK handshake used in BISYNC transmission after the receiver has checked to see if the data was received correctly. For electronic mail, the message can be written to the addressed mailbox and then read back to make sure it was sent correctly.

The *network layer* of the protocol is used only in multichannel networks. It is responsible for finding a path through the network to the desired receiver by switching between channels. The function of this layer is similar to the function of postal mail routing, which finds a route to get a letter from your house to the addressed destination. Another example of the function performed by this layer is the telephone switching system, which finds a route to connect a phone call.

The *data link layer* of the OSI model is responsible for the transmission of packets or blocks from sender to receiver. At this level the BCC characters or CRC characters are generated and checked, zeros are stuffed in the data, and flags and addresses are added to data frames. The HDLC data transmission protocol described earlier in this chapter is an example of the type of factors involved in this layer.

The *physical layer* of the OSI model is the lowest level. This layer is used to specify the connectors, cables, voltage levels, bit rates, modulation methods, etc. RS-232C is an example of a standard which falls in this layer of the model.

We don't have space here to discuss all the different networks listed as examples in Figure 14-33, but we will discuss a few of the most common ones. To start we will take a more detailed look at the operation of a very widespread "common-bus" network, Ethernet. Ethernet is a trademark of Xerox Corporation.

Ethernet

The *Ethernet network standard* was originally developed by Xerox Corporation. Later Xerox, DEC, and Intel worked on defining the standard sufficiently so that commercial products for implementing the standard were possible. It has now been adopted, with slight changes, as the IEEE 802.3 standard.

Physically, Ethernet is implemented in a common-bus topology with a single 50-Ω coaxial cable. Data is sent over the cable using baseband transmission at 10 Mbits/s. Data bits are encoded using Manchester coding, as shown in Figure 14-35. The advantage of this coding is that each bit cell contains a signal transition. A system that wants to transmit data on the network first checks for these transitions to see if the network is currently busy. If the system detects no transitions, then it can go ahead and transmit on the network.

Figure 14-36 shows how a very simple Ethernet is set up. The backbone of the system is the coaxial cable. Terminations are put on each end of the cable to prevent signal reflections and each unit is connected into the cable with a simple tee-type tap. A transmitter-receiver, or *transceiver*, sends out data on the coax, receives data from the coax, and detects any attempt to transmit while the coax is already in use. The transceiver is connected to an interface board with a 15-pin connector and four twisted-wire pairs. The transceiver cable can be as long as 15 m. The interface board, as the name implies, performs most of the work of getting data on and off the network in the correct form. The *interface board* assembles and disassembles data frames, sends out source and destination addresses, detects transmission errors, and prevents transmission while some other unit on the network is transmitting.

The method used by a unit to gain access to the network is *CSMA/CD*. Before a unit attempts to transmit on the network, it looks at the coax to see if a carrier (Manchester code transitions) is present. If a carrier is present, the unit waits some random length of time and then tries again. When the unit finds no carrier on the line, it starts transmitting. While it is transmitting, it also monitors the line to make sure no other unit is transmitting at the same time. The question may occur to you at this point, If a unit cannot start transmitting until it finds no carrier on the coax, how can another unit be transmitting at the same time? The answer to this question involves propagation delay. Since transceivers can be as much as 2500 m apart, it may take as long as 23 μs for data transmitted from one unit to reach another unit. In other words, one unit may start transmitting before the signal from a transmitter that started earlier reaches it. A situation where two units transmit at the same time is referred to as a *collision*. When a unit detects a collision, it will keep transmitting until all transmitting stations detect that a collision

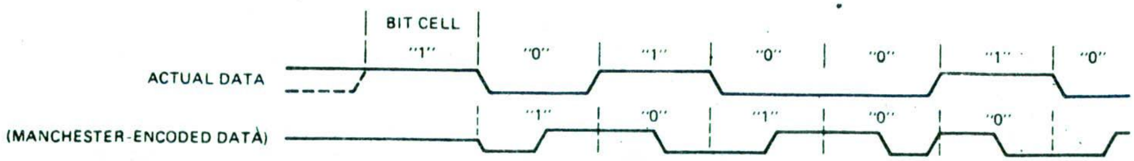


FIGURE 14-35 Manchester coding used for Ethernet data communication. Note that encoded data has a transition at center of each bit time.

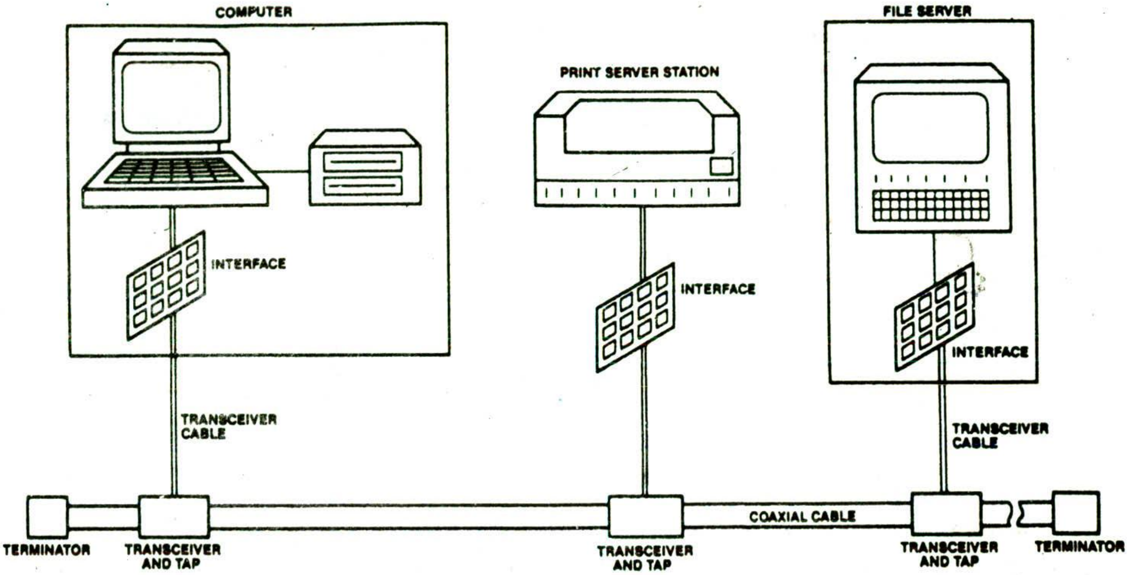


FIGURE 14-36 Block diagram of very simple Ethernet system. *(Intel Corporation)*

has occurred and then stop transmitting. Any other transmitting units will also stop transmitting and try again after a random period of time. The term "multiple access" in the CSMA/CD name means that any unit on the network can attempt to transmit. The network has no central controller to control which unit has use of the network at a particular time. Access is gained by any unit using the mechanism we have just described.

The maximum number of units that can be connected on a single Ethernet is 1024. For further information about how an interface board is built, consult the data sheets for the Intel 82586 LAN coprocessor and the data sheets for the Intel 82501 Ethernet serial interface.

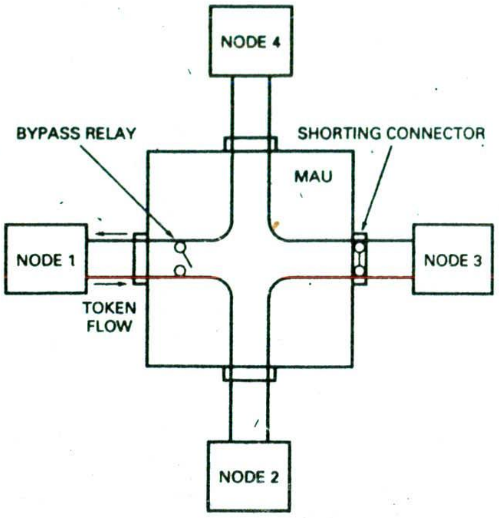
One problem with standard Ethernet is the coax cable used to connect units on the network. This cable is expensive and somewhat difficult to get through wiring conduits in existing buildings. To solve these problems, a new Ethernet standard called *thin Ethernet* or *10BaseT* was developed. The 10 in this name indicates 10-M bits/s transmission and the T in the name stands for twisted-pair telephone wire. By limiting the maximum distance between units to 100m rather than the 2500-m maximum for standard Ethernet, a 10BaseT network can use standard telephone-type wiring, which is often already installed or can easily be installed. The basic operation of the 10BaseT network is basically the same as that of the standard Ethernet we described previously.

Another problem with Ethernet is that as the amount of traffic on the network increases, the time that a unit on the end of the network has to wait before it can transmit may become very long. As the number of units increases, the number of collisions and the amount of time spent waiting for a "clear shot" increases. This degrades the performance of the network. In the next section we discuss token-passing ring networks, which solve the access problem in a way which degrades less under heavy traffic load.

Token-Passing Rings

IEEE standard 802.5 defines the physical layer and the data link layer for a *token-passing ring network*. As the name implies, systems on a token-passing ring are connected in series around a ring. To simplify wiring, however, token rings are often connected as shown in Figure 14-37, page 526. The multistation access unit or MAU is put in a wiring closet or some readily accessible place. Unlike the passive taps used in an Ethernet system, each active station or node on a token ring receives data, examines it to see if the data is addressed to it, and retransmits the data to the next station on the ring. A bypass relay in the MAU will automatically shunt data around defective or inactive nodes. Data always travels in one direction around the ring. Data is transmitted as HDLC or SDLC frames. Early token-ring network adapter cards transmitted data at 4 Mbits/s, but 16-Mbits/s network adapter cards are now becoming widely available.

Token-passing ring networks solve the multiple-access problem in an entirely different way from the CSMA/CD approach described for Ethernet. A token is a byte of data with an agreed-upon, unique bit pattern such as 01111111. If no station is transmitting, this token is circulated continuously around the ring. When a station needs to transmit, it withdraws the not-busy token, changes it to a busy token of perhaps 01111110, and sends the busy token on around the ring. The transmitting unit then sends a frame of data around the ring to the intended receiver(s). When the transmitting



NOTE: MAU = MULTISTATION ACCESS UNIT

FIGURE 14-37 Block diagram of a token ring network system showing multistation access unit (MAU).

station receives the busy token and the data frame back again, it reads them in and removes them from the ring. It then sends out the not-busy token again. As soon as a transmitting station sends out the not-busy token again, the next station on the loop can grab the token and transmit on the network. The first station that transmitted cannot transmit again until the not-busy token works its way around the ring. This gives all units on the network a chance to transmit in a "round-robin" manner.

> NOTE: Some token-ring networks use tokens with priority bits so that a high-priority station can transmit again if necessary before a lower-priority station gets a turn.

Two questions occurred to us the first time we read about token-passing rings; perhaps these same two questions may have occurred to you. The first question is, How does a station on the network tell the bit pattern for a token from the same bit pattern in the data frame? The answer to this question is bit-stuffing, the same technique that is used to prevent the flag bit pattern from being present in the data section of an HDLC frame. A hardware circuit in the transmitter alters the data stream so that certain bit patterns are not present. Another hardware circuit in the receiver reconstructs the original data.

The second question is, What happens if the not-busy token somehow gets lost going around the ring? A couple of different approaches are used to solve this problem. One approach uses a timer in each station. When a station has a frame to transmit, it starts a timer. If the station does not detect a token in the data stream before the timer counts down, it assumes that the token was lost and sends out a new token. Another approach used by IBM sets up one station as a network monitor. If this station does not detect a token within a prescribed time, it clears any leftover data from the ring and sends out a new not-busy token.

The Texas Instruments TMS380 chip set can be used to implement a node on a 4-Mbits/s token-ring network. Consult the data sheets for these devices to get more information about the operation of a token-ring network.

Token-passing ring networks have the disadvantage that more complex hardware is required where each station connects to the network, but as we said earlier, under heavy traffic loads they are more efficient than Ethernets. Also, the receive and transmit circuitry at the connection acts as a repeater, which helps maintain signal quality throughout the network. Since signals travel in only one direction around the ring, this topology is ideally suited for fiber-optic transmission.

A new standard called the Fiber Distributed Data Interface (FDDI) or ANSI X3T9.5 describes a fiber-optic token ring network which transmits data at 100 Mbits/s. The FDDI ring actually consists of a fiber which transmits data in one direction around the ring and another fiber which transmits data in the other direction around the ring. This dual-fiber approach allows data transmission to continue if one fiber path is broken or interrupted in some way. Nodes on FDDI can be as far as 2 km from each other, up to 500 nodes can be connected on the ring, and the maximum circumference of the ring can be much as 100 km. The Advanced Micro Devices' Supernet chip set or the National Semiconductor FDDI chipset can be used along with a microcontroller, buffer memory, and an electro-optical interface to build an FDDI node. Consult the data sheets for these devices to get more information about FDDI operation.

Figure 14-38 shows how an FDDI network can serve as a *backbone* which allows high speed communication between other networks. Circuits called *bridges* or *gateways* interface Ethernets, multiplexed Ethernets, or even T1 type signals with the FDDI. The Pentagon uses a network such as this.

A transmission rate of 100 Mbits/s may at first seem like "overkill," but as we move more and more toward high-resolution interactive video, computer simulations, and massive data storage, this rate is not nearly fast enough. Work is currently underway on fiber-optic networks that transmit data at 250 Mbits/s and 500 Mbits/s and allow nodes to be as much as 50 km apart.

A Network Application Example and LAN Software Overview

As an example of how you put all the pieces of a network together, suppose that you have the job of designing and setting up a general purpose computer room at a college. The lab is to be used for computer-aided drafting (CAD) with AutoCAD; programming in Pascal, C and assembly language; mechanical engineering simulations; word processing; and other unspecified applications. All programs that will be run require an IBM PC- or PS/2-type computer. The computer room is to have 24 workstations, a large plotter, a laser printer, and two letter-quality dot-matrix printers.

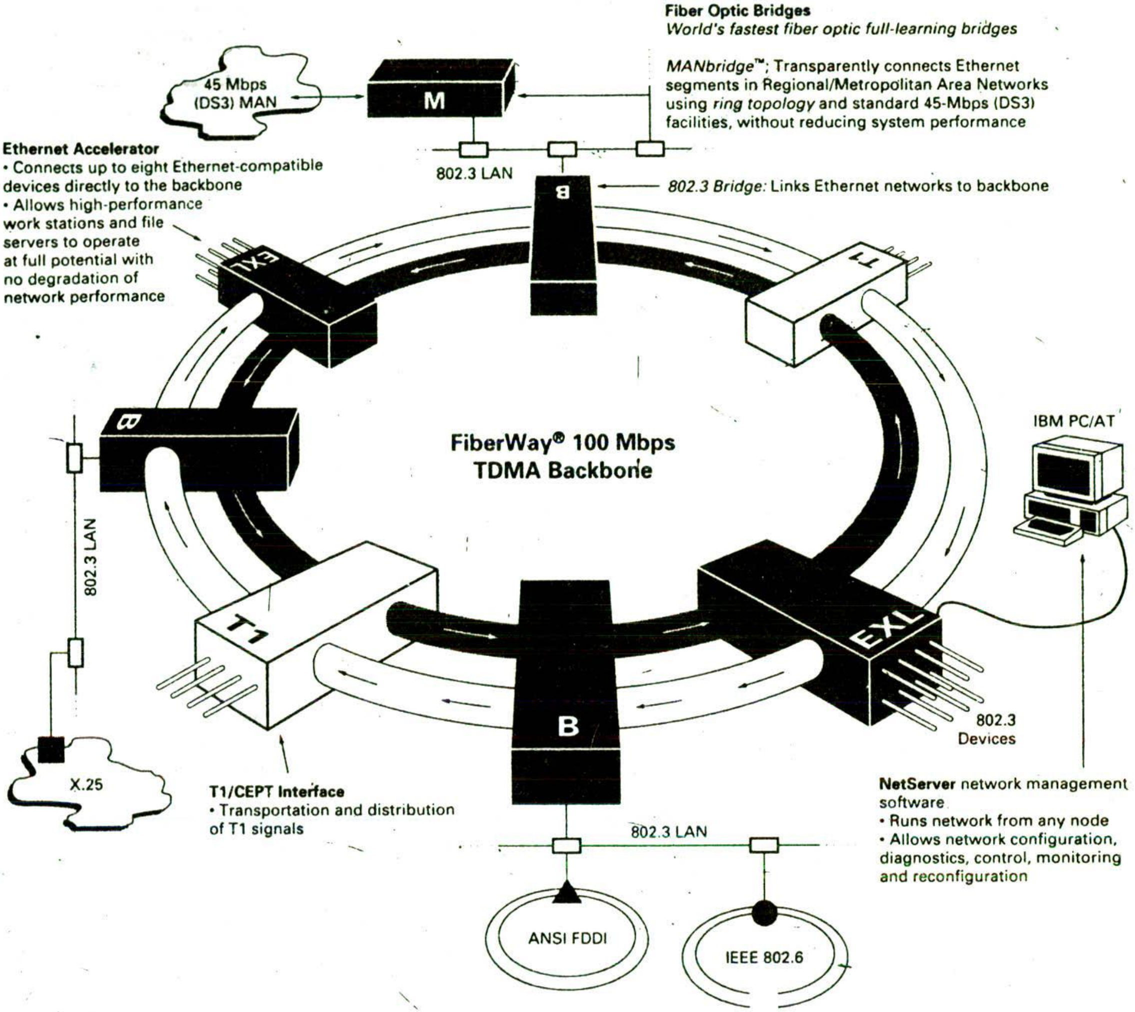

Fiber Optic Bridges
World's fastest fiber optic full-learning bridges
MANbridge™; Transparently connects Ethernet segments in Regional/Metropolitan Area Networks using ring topology and standard 45-Mbps (DS3) facilities, without reducing system performance
45 Mbps (DS3) MAN
M
802.3 LAN
Ethernet Accelerator
• Connects up to eight Ethernet-compatible devices directly to the backbone
• Allows high-performance work stations and file servers to operate at full potential with no degradation of network performance
B
802.3 Bridge: Links Ethernet networks to backbone
EXL
T1
B
FiberWay® 100 Mbps TDMA Backbone
IBM PC/AT
802.3 LAN
T1
EXL
B
802.3 Devices
X.25
T1/CEPT Interface
• Transportation and distribution of T1 signals
802.3 LAN
NetServer network management software
• Runs network from any node
• Allows network configuration, diagnostics, control, monitoring and reconfiguration
ANSI FDDI
IEEE 802.6

FIGURE 14-38 Fiber distributed data interface (FDDI) network used as "backbone" for different types of networks. (*ARTEL Communications.*)

The drafting and mechanical engineering instructors indicate that they need the speed of an 80386-based machine and display resolution of 1024 × 768 pixels. These specifications require that each of 24 workstations be an 80386-based machine with an 8514/A-type video adapter. Some of the programs that they plan to run are very memory hungry, so the basic workstations need 2 Mbytes or more of RAM.

The systems need to run a very wide variety of programs, so a large amount of hard-disk storage is needed. One alternative is to install a large hard disk in each workstation and install a set of the required programs on each disk. One problem with this approach is the cost of the 24 large hard disks. A second problem with this approach is that it is difficult to maintain the software on all these separate machines. Updating is tedious and time-consuming. Still another problem is that on these individual machines it is difficult to protect the application programs from accidental or mischievous corruption by users.

All these problems can be solved by connecting the workstations on a network which includes a fast file server. A single copy of the application programs can be installed on the file server and accessed from each workstation as needed. If the hard disk on the server is large enough, user files can also be stored on it. The plotter and printers can also be connected on the file server so that they are accessible from any workstation.

The file server and its hard disk need to be fast so that they do not create a bottleneck in the system. You might choose an 80486-based microcomputer for the file server and equip it with a 250-Mbyte, 16-ms hard disk. If your budget permits, you might also include an optical disk drive in the server so that programming classes could access the Microsoft Programmer's Library which is available on CD ROM. The server will also need a 1.2-Mbyte floppy drive and a 1.44-Mbyte floppy drive to transfer software from floppies to the hard disk.

The next step is to decide on the software you want to use to manage the network and to provide the file server and print server functions. The best approach for this is to choose the network software which will do the best job and then choose network hardware which is compatible with that software. As we write this discussion, the best choice seems to be Novell's Netware 386, so we will use it as an example.

Netware 386 works with Ethernet, ARCnet, and IBM's Token Ring boards. Since the workstations in this lab are physically all in the same room, you might consider using the 10BaseT or thin ethernet network we described earlier, because it is the cheapest of these alternatives. Remember that this type network transmits data at 10 Mbits/s over standard twisted-pair phone wire for distances up to 100 m. Synoptics, 3Com, and several other companies make adapter boards which interface PC or PS/2 buses to a 10BaseT network.

Netware requires a minimum 2 Mbytes of memory in the server, and it works better with 8M or 10M, so you should include this in the bid specifications for the server.

While you are waiting for hardware bids to come in, purchase orders to go out, and the hardware to arrive, we will give you an overview of how network software works so you will have some idea how to install and use it.

Part of the network software resides in each workstation and part of it resides in the server. Let's start with the workstation part. To refresh your memory, Figure 14-39*a* shows the software hierarchy for a DOS-based workstation operating in stand-alone mode. In this mode an application program such as a word processor uses DOS function calls to access system peripherals. The DOS function calis use BIOS procedures to interact with the actual hardware.

Figure 14-39*b* shows the software hierarchy when the workstation is operating in network mode. When the application program attempts to access a disk file, for example, the "interceptor" part of the resident network software determines whether the file is located on the workstation hard disk or on the server hard disk. If the file is on the workstation hard disk, the interceptor simply passes the request on to DOS and the access proceeds through DOS and BIOS as it would in stand-alone operation. If the file is on the server, the request goes to the request translator to get assembled in the proper packet format for transmission over Ethernet. The output from the request translator then goes to the network communications driver, which sends it to the server over the network. A standard set of network communication drivers written by Microsoft is called *NETBIOS*. Other companies which write network control programs either license NETBIOS from Microsoft or write their own compatible network drivers.

The server reads the requested file, converts it to packets, and sends it to the workstation. The appropriate driver reads the packets into the workstation.

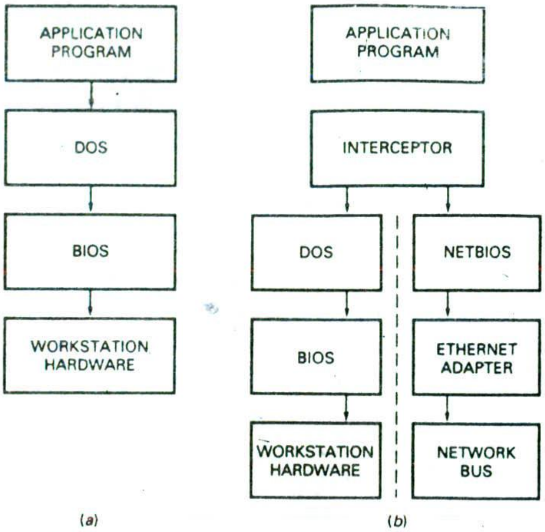


FIGURE 14-39 Software hierarchy on a workstation. (*a*) Nonnetworked. (*b*) Networked.

The reply translator part of the software converts the packets to DOS file format and loads the file in memory so the application can work with it.

The network software that resides in the server is a complete operating system in itself. To install Netware 386 on the server, you first format a small partition on the server hard disk in DOS format. You then load DOS in this partition so the system is bootable from it and load some of the basic Netware files here so you can install the rest of Netware. After you boot the system, the installation consists of working your way through a relatively simple sequence of steps outlined in the installation procedure.

Once installed, the network operating system is set up so that only the system administrator can access and change its operation. The system administrator sets up user accounts, assigns passwords, and sets the access rights for files. Application program files are usually specified as read-only so that users cannot accidentally or maliciously modify them. For files that are intended to be accessed and written to by any one of several users, Netware 386 has a default feature called *file locking*, which prevents one user from accessing the file until a previous user has finished with it. In this case the critical region is the file, and file locking provides a way to protect it.

Netware 386 uses several techniques to speed up disk access. First, it formats its partition of the hard disk differently from the way DOS formats it to make for more efficient access to the parts of a file. Second, it uses disk caches such as those we described in the last chapter to hold large blocks of data read from files. This reduces the number of read operations required to access a large file. Finally, Netware uses "elevator seeking" to reduce the amount the heads move to read a requested set of files for users. Just as an elevator moves sequentially from floor to floor instead of moving from floor to floor as requested, the head is moved to access files in the sequence they are located on disk tracks rather than strictly in the sequence they were requested.

In addition to allowing users to store and access files, the network operating system has many other useful features. It sets up a queue of files waiting to be printed or plotted so that users can just enter a print command and go on with their work. Most networks have electronic mail, which allows the system administrator to communicate with all users and users to communicate with each other. Most electronic mail systems are set up so you can define a group of users and direct mail messages to just that group.

For the reasons that we have discussed, it is likely that in the near future almost all computers will in some way be networked with other computers through telephone lines or direct connections. In the last section of the chapter we discuss a different type of computer network which is often used in a factory environment to build a "smart" test system.

THE GPIB, HPIB, IEEE488 BUS

The preceding sections of the chapter discussed networks which allow microcomputers to communicate with each other and to share peripherals such as printers. The *general-purpose interface bus* (GPIB), also known as the *Hewlett-Packard interface bus* and the *IEEE488 bus* that we discuss here is not intended for use as a computer network in the same way that the Ethernet and token rings are used. It was developed by Hewlett-Packard to interface smart test instruments with a computer.

The standard describes three types of devices that can be connected on the GPIB. First is a *listener*, which can receive data from other instruments or from the controller. Examples of listeners are printers, display devices, programmable power supplies, and programmable signal generators. The second type of device defined is a *talker*, which can send data to other instruments. Examples of talkers are tape readers, digital voltmeters, frequency counters, and other measuring equipment. A device can be both a talker and a listener. The third type of device on the bus is a *controller*, which determines who talks and who listens on the bus.

Physically the bus consists of a 24-wire cable with a connector such as that shown in Figure 14-40*a*, page 530, on each end. Actually, each end of the cable has both a male connector and a female connector, so that cables can daisy-chain from one unit to the next on the bus. Instruments intended for use on a GPIB usually have some switches which allow you to select the 5-bit address that the instrument will have on the bus. Standard TTL signal voltage levels are used.

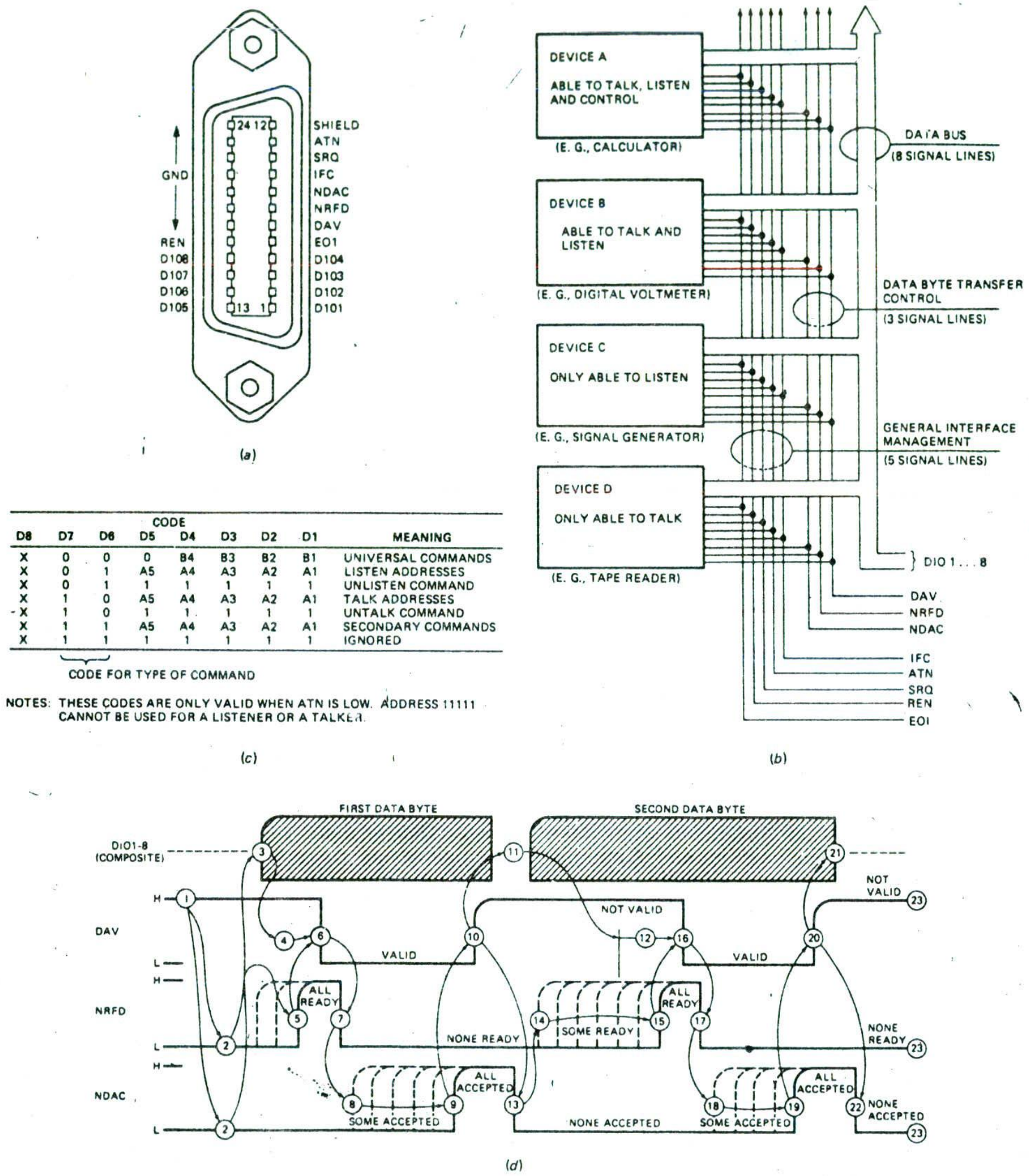
As shown in Figure 14-40*b*, the GPIB has eight bidirectional data lines. These lines are used to transfer data, addresses, commands, and status bytes among as many as 8 or 10 instruments.

The GPIB also has five bus management lines which function basically as follows. The *interface clear* line (IFC), when asserted by the controller, resets all devices on the bus to a starting state. It is essentially a system reset. The *attention* (ATN) line, when asserted (low), indicates that the controller is putting a universal command or an address-command such as "listen" on the data bus. When the ATN line is high, the data lines contain data or a status byte. *Service request* (SRQ) is similar to an interrupt. Any device that needs to transfer data on the bus asserts the SRQ line low. The controller then polls all the devices to determine which one needs service. When asserted by the system controller, the *remote enable* (REN) signal allows an instrument to be controlled directly by the controller rather than by its front-panel switches. The *end or identify* (EOI) signal is usually asserted by a talker to indicate that the transfer of a block of data is complete.

Finally, the bus has three handshake lines that coordinate the transfer of data bytes on the data bus. These are *data valid* (DAV), *not ready for data* (NRFD), and *not data accepted* (NDAC). These handshake signals allow devices with very different data rates to be connected together in a system. A little later we will show you how this handshake works. First we will give you an overview of general bus operation.

Upon power-up the controller takes control of the bus and sends out an IFC signal to set all instruments on the bus to a known state. The controller then proceeds to use the bus to perform the desired series of measurements or tests. To do this the controller sends out a series of commands with the ATN line asserted low. Figure 14-40c shows the formats for the combination command-address codes that a controller can send to talkers and listeners. Bit 8 of these words is a don't care, bits 7 and 6 specify which command is being sent, and bits 5 through 1 give the address of the talker or listener to which the command is being sent. For example, to enable (address) a device at address 04 as a talker, the controller simply asserts the ATN line low and sends out a command-address byte of X1000100 on the data bus. A listener is enabled by sending out a command-address byte of $X01A_5A_4A_3A_2A_1$, where the lower 5 bits contain the address that the listener has been given in the system. When a data transfer is complete, all listeners are turned off by the controller sending an unlisten command, X0111111. The controller turns off the talker by sending an untalk command, X1011111. *Universal commands* sent by the controller with bits 7, 6, and 5 all 0's will go to all listeners and talkers. The lower 4 bits of these words specify one of 16 universal commands.

Periodically while it is using the bus, the controller checks the SRQ line for a service request. If the SRQ line is low, the controller polls each device on the bus one after another (serial) or all at once (parallel) until it finds the device requesting service. A talker such as a DVM, for example, might be indicating that it has completed a series of conversions and has some data to send to a listener such as a chart recorder. When the controller determines the source of the SRQ, it asserts the ATN line low and sends listener address commands to each listener that is to receive the data and a talk address command to the talker that requested service. The controller then raises the ATN line high, and data



			CODE					
D8	D7	D6	D5	D4	D3	D2	D1	MEANING
X	0	0	0	B4	B3	B2	B1	UNIVERSAL COMMANDS
X	0	1	A5	A4	A3	A2	A1	LISTEN ADDRESSES
X	0	1	1	1	1	1	1	UNLISTEN COMMAND
X	1	0	A5	A4	A3	A2	A1	TALK ADDRESSES
X	1	0	1	1	1	1	1	UNTALK COMMAND
X	1	1	A5	A4	A3	A2	A1	SECONDARY COMMANDS
X	1	1	1	1	1	1	1	IGNORED

CODE FOR TYPE OF COMMAND

NOTES: THESE CODES ARE ONLY VALID WHEN ATN IS LOW. ADDRESS 11111 CANNOT BE USED FOR A LISTENER OR A TALKER.

FIGURE 14-40 GPIB pins, signals, and handshake waveforms. (*a*) Connector. (*b*) Bus structure. (*c*) Command formats. (*d*) Data transfer handshake waveforms.

is transferred directly from the talker to the listeners using a double-handshake-signal sequence.

Figure 14-40*d* shows the sequence of signals on the handshake lines for a transfer of data from a talker to several listeners. The DAV, NRFD, and NDAC lines are all open-collector. Therefore, any listener can hold NRFD low to indicate that it is not ready for data or hold NDAC low to indicate that it has not yet accepted a data byte. The sequence proceeds as follows. When all listeners have released the NRFD line (5 in Figure 14-40*d*), indicating that they are ready (not not-ready), the talker asserts the DAV line low to indicate that a valid data byte is on the bus. The addressed listeners then all pull NRFD low and start accepting the data. When the slowest listener has accepted the data, the NDAC line will be released high (9 in Figure 14-40*d*). The talker senses NDAC becoming high and unasserts its DAV signal. The listeners pull NDAC low again, and the sequence is repeated until the talker has sent all the data bytes it has to send. The rate of data transfer is determined by the rate at which the slowest listener can accept the data.

When the data transfer is complete, the talker pulls the EOI line in the management group low to tell the controller that the transfer is complete. The controller then takes control again and sends an untalk command to the talker. It also sends an unlisten command to turn off the listeners and continues to use the bus according to its internal program.

A standard microprocessor bus can be interfaced to the GPIB with dedicated devices such as the Intel 8291 GPIB talker-listener and 8292 GPIB controller. The importance of the GPIB is that it allows a microcomputer to be connected with several test instruments to form an integrated test system.

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

Serial-data communication
  Simplex, half-duplex, full-duplex
  Synchronous, asynchronous
  Marking state, spacing state
  Start bit, stop bit
  Baud rate

UART, USART, DTE, DCE

20- and 60-mA current loops

RS-232C, RS-422A, RS-423A, RS-449 serial-data standards

CODECs, TDM, and PCM

ISDN

Modems

Amplitude Modulation, FSK, PSK
  Quaternary amplitude modulation (QAM)
  Scrambler, descrambler

Fiber-optic data communication
  Critical angle
  Multimode and single-mode fibers

Terminal emulator

Circular buffer, ring buffer
  Head pointer, tail pointer

Critical region

Binary synchronous communications protocol (BISYNC)
  Byte-controlled protocol (BCP)
  Cyclic redundancy check
  XMODEM protocol

HDLC, SDLC protocols
  Bit-oriented protocol (BOP)
  Frame, field, flag
  Frame check sequence (FCS)

Local area network (LAN)

Topologies—star, loop, ring, common-bus, broadband-bus (tree)

Electronic mail

Open system interconnection model (OSI)
  Presentation, session, transport, network, data link
  Physical layers

Ethernet
  Transceiver
  Collision
  CSMA/CD
  10 BaseT

Token-passing rings

Fiber distributed data interface (FDDI)

File server, print server

GPIB, HPIB, IEEE 488 bus standard
  Listener, talker, controller

REVIEW QUESTIONS AND PROBLEMS

1. Draw a diagram showing the bit format used for asynchronous serial data. Label the start, stop, and parity bits. Number the data bits to show the order of transmission.

2. A terminal is transmitting simple asynchronous serial data at 1200 Bd.
   a. How much time is required to transmit 1 bit?
   b. Assuming 7 data bits, a parity bit, and 1 stop bit, how long does it take to transmit one character?

3. What is the main difference between a UART and a USART?

4. Define the term *modem* and explain why a modem is required to send digital data over standard switched phone lines.

5. Describe the functions of the $\overline{\text{DSR}}$, $\overline{\text{DTR}}$, $\overline{\text{RTS}}$, $\overline{\text{CTS}}$, TxD, and RxD signals exchanged between a terminal and a modem.

6. What frequency transmit clock (TxC) is required by an 8251A in order for it to transmit data at 4800 Bd with a baud rate factor of 16?

7. *a.* Show the bit pattern for the mode word and the command word that must be sent to an 8251A to initialize the device as follows: baud rate factor of 64, 7 bits/character, even parity, 1 stop bit, transmit interrupt enabled, receive interrupt enabled, $\overline{DTR}$ and $\overline{RTS}$ asserted, error flags reset, no hunt mode, no break character.
   b. Show the sequence of instructions required to initialize an 8251A at addresses 80H and 81H with the mode and command words you worked out in part *a*.
   c. Show the sequence of instructions that can be used to poll this 8251A to determine when the receiver buffer has a character ready to be read.
   d. How can you determine whether a character received by an 8251A contains a parity error?
   e. What frequency transmit and receive clock will this 8251A require in order to send data at 2400 Bd?
   f. What other way besides polling does the 8251A provide for determining when a character can be sent to the device for transmission? Describe the additional hardware connections required for this method.

8. Give the signal voltage ranges for a logic high and for a logic low in the RS-232C standard.

9. *a.* Describe the problem that occurs when you attempt to connect together two RS-232C devices that are both configured as DTE.
   b. Draw a diagram which shows how this problem can be solved.

10. *a.* Why are the two ground pins on an RS-232C connector not just jumpered together?
    b. What symptom will you observe if the wire connected to pin 5 of an RS-232C terminal is broken?

11. Explain why systems which use the RS-422A or RS-423A signal standards can transmit data over longer distances and at higher baud rates than RS-232C systems.

12. *a.* How does an FSK modem represent digital 1's and 0's in the signal it sends out on a phone line?
    b. How does an FSK modem perform full-duplex communication over standard phone lines?
    c. Approximately what is the maximum bit rate for FSK data transmission on standard switched telephone lines?

13. *a.* Draw a waveform to show the signal that a simple phase-shift keying (PSK) modem will send out to represent the binary data 011010100.
    b. Describe how phase shift modulation can be used to transmit 2 data bits with only one carrier change.
    c. Describe how quatenary amplitude modulation transmits 4 data bits with only one carrier change.

14. *a.* Why do telephone companies transmit signals over long distances in digital form rather than in analog form?
    b. Describe the operation of a codec.
    c. Why are codecs designed with nonlinear response?
    d. Explain how telephone companies commonly transmit many phone signals on a single wire or channel.

15. *a.* Briefly describe the operation of the integrated services digital network.
    b. Explain the significance ISDN has for data communication between computers.

16. *a.* Draw a diagram which shows the construction of a fiber-optic cable, and label each part.
    b. Identify two types of devices which are used to produce the light beam for a fiber-optic cable and two devices which are commonly used to detect the light at the receiving end of the fiber.
    c. Why should you never look into the end of a fiber optic cable to see if light is getting through?
    d. Describe the difference between a multimode fiber and a single-mode fiber. Give a major advantage and a major disadvantage of each type.
    e. What are the major advantages of fiber-optic cables over metallic conductors?

17. Using IBM PC BIOS and DOS calls, write an assembly language program which reads characters from the keyboard and puts them in a buffer until a carriage return is entered. The characters should be displayed on the CRT as entered. When a carriage return is entered, the contents of the buffer should be sent out the COM1 serial port.

18. The SDK-86 will accept only uppercase letters as commands. The SDK-86 emulator program in Figure 14-25 would be friendlier if you did not have to remember to press the caps lock key on the IBM. Write an assembly language routine that will convert a letter entered in lowercase to uppercase without affecting entered uppercase letters or numbers and describe where you would insert this section of code in the program in Figure 14-25.

19. Describe the operation of a circular or ring buffer. Include in your answer the function of the tail pointer, the head pointer, and how the buffer-full and buffer-empty conditions are detected.

20. Why is it necessary to disable the UART interrupt input of the 8259A during part of the CHK_N_DISPLAY procedure in Figure 14-27*b*?

21. *a.* When changing a bit in a control word or interrupt mask word, why should you not alter the other bits in the word?

b. Show the assembly language instructions you would use to unmask IR5 of an 8259A at base address 80H without changing the interrupt status of any other bits.

22. Why is synchronous serial data communication much more efficient than asynchronous communication?

23. *a.* If an 8251A is being used in synchronous mode for a BISYNC data link, what additional initialization word(s) must be sent to the device?

b. How does the 8251A detect the start of a message?

c. How does the 8251A indicate that it has found the start of a message?

d. How does the receiving station in a BISYNC link indicate that it found an error in the received data?

24. *a.* How is the start of a message frame indicated in a bit-oriented protocol such as HDLC?

b. How does an HDLC system prevent the flag bit pattern from appearing in the data part of the message?

c. How does the receiver in an HDLC system tell the transmitter that an error was found in a transmitted frame?

25. *a.* Draw simple diagrams which show the five common network topologies.

b. For each topology identify one commercially available system which uses it.

26. What is the difference between a baseband network and a broadband network?

27. *a.* List the seven layers of the ISO open systems model.

b. Which of these layers is responsible for assembling messages into frames or packets?

c. Which layer is responsible for making sure the message was transmitted and received correctly?

28. *a.* Describe the topology, physical connections, and signal type used in Ethernet.

b. Describe the method used by a unit on an Ethernet to gain access to the network for transmitting a message.

c. What response will a transmitting station make if it finds that another station starts transmitting after it starts?

d. What is the term used to refer to this condition?

29. *a.* Describe the method used by a unit on a token-passing ring to take control of the network for transmitting a message frame.

b. What is the advantage of this scheme over the method used in Ethernet?

c. How can a token ring network recover if the token is lost while being passed around the ring?

30. *a.* Describe how the software on a network node responds when the user enters a command which accesses the hard disk in the workstation.

b. Describe how the software on a network node responds when the user enters a command which accesses the hard disk on the file server.

c. Describe how the file server software protects application program files from being modified by users.

d. Describe how the file server software protects user files from access by other users.

31. *a.* For what purpose was the GPIB designed?

b. Give the names for the three types of devices which the GPIB defines.

c. List and briefly describe the function of the three signal groups of the GPIB.

d. Describe the sequence of handshake signals that take place when a talker on a GPIB transfers data to several listeners. How does this handshake scheme make it possible for talkers and listeners with very different data rates to operate correctly on the bus?

CHAPTER 15

The 80286, 80386, and 80486 Microprocessors

For most of the examples up to this point in the book, we have used the 8086/8088 microprocessor, because it is the simplest member of this family of Intel processors and is therefore a good starting point. Now it is time to look at the evolutionary offspring of the 8086. To give you an overview, here are a few brief notes about the members of this family.

The 80186 processor is basically an 8086 with an on-chip priority-interrupt controller, programmable timer, DMA controller, and address decoding circuitry. This processor has been used mostly in industrial control applications.

The 80286, another 16-bit enhancement of the 8086, was introduced at the same time as the 80186. Instead of the integrated peripherals of the 80186, it has virtual memory-management circuitry, protection circuitry, and a 16-Mbyte addressing capability. The 80286 was the first family member designed specifically for use as the CPU in a multiuser microcomputer.

The 80386, the next evolutionary step in the family, is a 32-bit processor with a 32-bit address bus. The 32-bit ALU allows the 80386 to process data faster, and the 32-bit address bus allows the 80386 to address up to 4 Gbytes of memory. Another enhancement of the 80386 is that segments can be as large as 4 Gbytes instead of only 64 Kbytes. The memory-management circuitry and protection circuitry in the 80386 are improved over that in the 80286, so the 80386 is much more versatile as the CPU in a multiuser system.

The latest current member of this family, the 80486, has the same CPU as the 80386, so it has the same addressing capability, memory-management, and protection features as the 80386. The main new features included in the 80486 are a built-in 8-Kbyte code/data cache and a 32-bit floating-point-unit, similar to the 8087 we discussed in Chapter 11.

As perhaps you can see from the preceding brief discussions, the 80286, 80386, and 80486 were designed for use as the CPU in a multitasking microcomputer system. To help you better understand the operation and design rationale of these processors, we start the chapter with a discussion of the problems that must be solved in writing a multitasking/multiuser operating system. We then discuss the 80286, 80386, and 80486 microprocessors in detail and explain how the features designed in these processors help solve the problems involved in implementing a multitasking operating system. After that we discuss how you develop real mode and protected mode programs for systems using these devices.

Finally in the chapter we discuss some of the directions in which microcomputer evolution seems to be heading. Included in this section are discussions of RISC processors, parallel processors, artificial intelligence, "fuzzy" logic, and neural networks.

OBJECTIVES

At the conclusion of this chapter, you should be able to:

1. Describe the difference between time-slice scheduling and preemptive priority-based scheduling.
2. Define the terms blocked, task queue, deadlock, deadly embrace, critical region, semaphore, kernel, memory-management unit, and virtual memory.
3. Describe how "expanded" memory is used to increase the amount of memory available in a microcomputer.
4. Describe how virtual memory gives a computer much more "logical" address space than the physical memory actually present in the system.
5. Describe the types of protection that should be implemented in a multitasking operating system.
6. Describe two methods that can be used to protect a critical region of code.
7. List the major hardware and software features that the 80286 microprocessor has beyond those in the 8086.
8. Show how the 80286 constructs physical addresses in its real address mode and in its protected virtual address mode.
9. List the evolutionary advances that the 80386 has over the 80286.
10. Describe how the 80386 produces a physical address when it is operating in paged mode.
11. Describe how segment-based protection is implemented in an 80386 system operating in protected mode.

12. Describe how an 80386 call gate is used to allow application programs to access operating systems procedures.

13. Describe how an 80386 performs a task switch.

14. Explain the term virtual 8086 mode for an 80386.

15. List the major advances that the 80486 has over the 80386.

16. Describe how system programs are developed for an 80386 or 80486 protected-mode system.

17. Describe how application programs are developed for 80386 or 80486 systems.

18. Describe the operation of the Microsoft Windows multitasking environment.

19. Define the terms RISC, CISC, artificial intelligence, expert system, neural network, and fuzzy logic.

MULTIUSER/MULTITASKING OPERATING SYSTEM CONCEPTS

Introduction

The basic principle of a timeshare system is that the CPU runs one user's program for a few milliseconds, then runs the next user's program for a few milliseconds, and so on until all of the users have had a turn. It cycles through the users over and over, fast enough that each user seems to have the complete attention of the CPU. An operating system which coordinates the actions of a timeshare system such as this is referred to as a *multiuser* operating system. The program or section of a program for each user is referred to as a *task* or *process*, so a multiuser operating system is also commonly referred to as *multitasking*. Multitasking operating systems are also used to control the operation of machines in industrial manufacturing environments. The factory controller program in Figure 10-35 is an example of a very simple multitasking operating system.

In this section we discuss some of the major problems encountered in building a multitasking operating system; then in later sections of the chapter we show you how the features of the 80286, 80386, and 80486 help solve these problems.

Scheduling

TSR PROGRAMS AND DOS

MS DOS is designed as a single-user, single-task operating system. This means that DOS can usually execute only one program at a time. The only exception to this in the basic DOS is the print program, print.com. You may have noticed that when you execute the print command, DOS returns a prompt and allows you to enter another command before the printing is completed. The print program starts printing the specified file and then returns execution to DOS. However, the print program continues to monitor DOS execution. When DOS is sitting in a loop waiting for a user command or some other event, the print program borrows the CPU for a short time and sends more data to the printer. It then returns execution to the interrupted DOS loop.

The DOS print command then is a limited form of multitasking. Products such as Borland's Sidekick use this same technique in DOS systems to provide pop-up menus of useful functions such as a calculator, an appointment book, and a notepad. The first time you run a program such as Sidekick, it is loaded into memory as other programs are. However, unlike other programs, Sidekick is designed so that when you terminate the program, it stays "resident" in memory. You can execute the program and pop up the menu again by simply pressing some hot key combination such as Ctrl-Alt. Programs which work in this way are called *terminate-and-stay-resident* or TSR programs. Because TSRs are so common in the PC world, we thought you might find it interesting to see how they work before we get into discussions of the scheduling techniques used in full-fledged multitasking operating systems.

When you boot up DOS, the basic 640 Kbytes of RAM are set up as shown in Figure 15-1*a*. Starting from absolute address 00000, the first section of RAM is reserved for interrupt vectors. The main part of the DOS program is loaded into the next-higher section of RAM. After this come device drivers such as ANSI.SYS, MOUSE.SYS, etc. The DOS command processor program, command.com, gets loaded into RAM at boot time. This program, which processes user commands and executes programs, has two parts. The resident part of the command processor is loaded in memory just above the device drivers and the transient part is loaded in at the very top of RAM. When you tell DOS to execute a

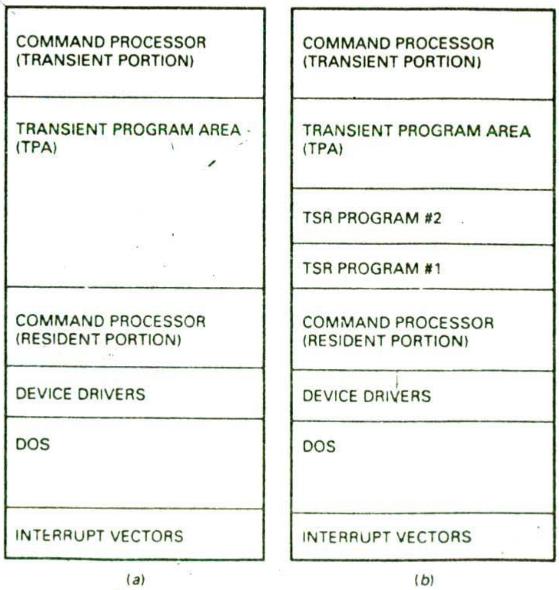

COMMAND PROCESSOR (TRANSIENT PORTION)
TRANSIENT PROGRAM AREA (TPA)
COMMAND PROCESSOR (RESIDENT PORTION)
DEVICE DRIVERS
DOS
INTERRUPT VECTORS
(a)
COMMAND PROCESSOR (TRANSIENT PORTION)
TRANSIENT PROGRAM AREA (TPA)
TSR PROGRAM #2
TSR PROGRAM #1
COMMAND PROCESSOR (RESIDENT PORTION)
DEVICE DRIVERS
DOS
INTERRUPT VECTORS
(b)

FIGURE 15-1 (*a*) DOS memory map without TSRs. (*b*) DOS memory map with TSRs.

.exe program, the program will be loaded into the transient program area of RAM and, if necessary, into the RAM where the transient part of the command processor was loaded. (The transient part of the command processor will be reloaded when the program terminates.)

Normally, when a program terminates all the transient program area is deallocated, so that another program can be loaded in it to be run. TSR programs, however, are terminated in a special way so that they are left resident in memory, as shown in Figure 15-1*b*. The transient program area is simply reduced by the size of the TSR program(s). When another program is loaded to be run, it is put in RAM above the TSRs.

One question that might occur to you at this point is, How do I make a program resident? To make the program stay resident when it terminates, you use the 31H subfunction of the DOS INT 21H function call. Specifically, you load AH with 31H, load AL with 00H, load DX with the length of the TSR program, and execute the INT 21H instruction. When the program is run from the command line or the AUTOEXEC.BAT file, it will be loaded into RAM, terminated, and left resident.

The next question that might occur to you then is, How does the TSR program get executed after it is resident? The answer to this question is that TSRs are executed as part of interrupt procedures. The exact mechanism depends on whether the TSR is active or passive.

An example of a passive TSR is the switch.com program which I use on my computer. The purpose of this program is to switch the functions of the Caps Lock key and the Ctrl key so that I don't have to retrain my finger to the key positions on my new keyboard. When DOS finds the statement "switch" in my AUTOEXEC.BAT file, it executes the switch.com program. The switch.com program terminates and remains resident. To accomplish the desired switch action, the program "intercepts" the BIOS keyboard interrupt, 09H, as shown in Figure 15-2*a*. You may remember that we showed you how to intercept interrupts at the start of the SDKCOM1 program in Figure 14-27. The result of this interception is that whenever a key is pressed, execution goes first to the switch TSR program. The switch program then calls the BIOS INT 09H procedure to read in codes from the keyboard. If the key code read from the keyboard represents a Caps Lock, it is replaced with the code for a Ctrl, and if the key code represents a Ctrl, it is replaced with the code for a Caps Lock. Other key codes are simply passed on as received. To DOS, then, the switch TSR is simply an interrupt procedure which is executed automatically when a key on the keyboard is pressed.

An example of an active TSR is Borland's Sidekick program, which pops up a menu of command options when you press the Ctrl key and the Alt key. As we mentioned before, Sidekick allows you to temporarily pause during some other program and write a note in a notebook file, perform a calculation on a screen-based calculator, check your appointment schedule, or any one of several other functions. To terminate Sidekick and return to the previously executing program, you press the Esc key. As with the passive switch.com TSR,

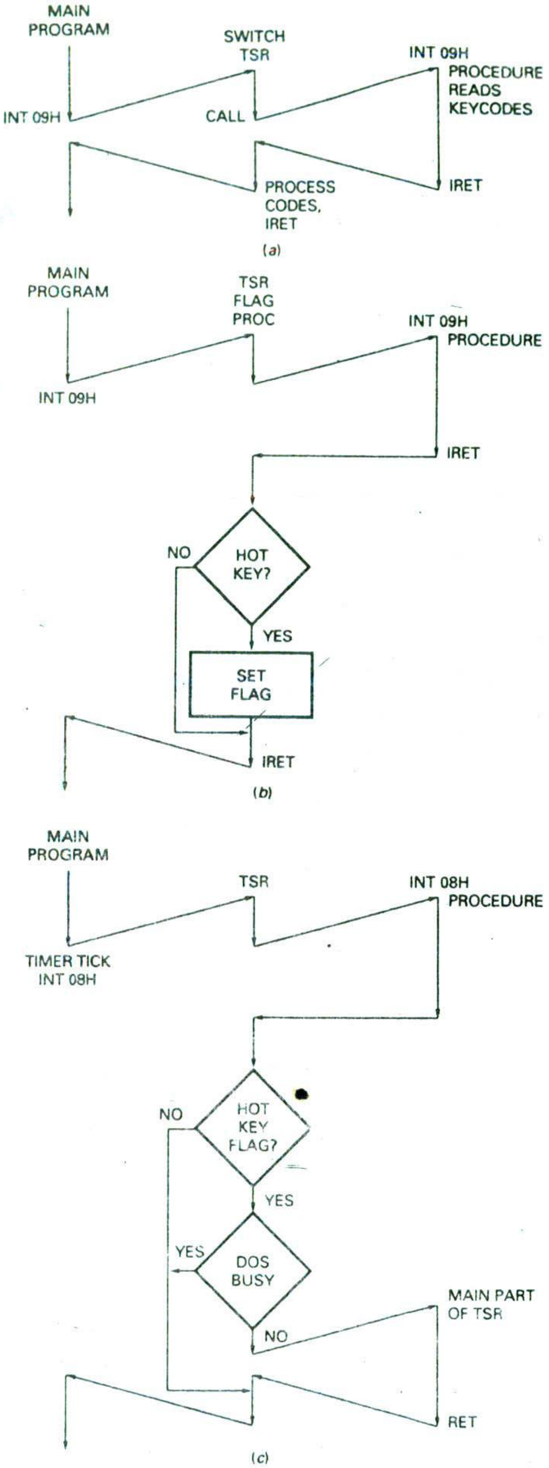


FIGURE 15-2 (*a*) Program flow for switch.com passive TSR. (*b*) Program flow for flag set part of active TSR. (*c*) Program flow for main part of TSR.

you make Sidekick resident by running it from the command line or as part of the AUTOEXEC.BAT file. Figure 15-2*b* and *c* show how an active TSR such as Sidekick is commonly executed when a hot key combination is pressed.

As shown in Figure 15-2*b*, the first part of the TSR intercepts the keyboard interrupt and immediately calls the BIOS keyboard procedure to read in the scan code from the keyboard. When execution returns from the BIOS INT 09H procedure, the TSR checks the returned key code to determine if a hot key was pressed. If a hot key was not pressed, execution simply returns to the interrupted program. If a hot key was pressed, the TSR procedure sets a global flag in memory before returning to the interrupted program. Another section of the TSR will check this flag at periodic intervals to determine if the main part of the TSR should be executed.

The part of the TSR which checks the hot key flag is often connected with the clock tick interrupt procedure, as shown in Figure 15-2*c*. The normal clock tick interrupt vector is replaced with the starting address of this section of the TSR. When a clock tick interrupt occurs (about every 18 ms for a PC- or PS/2-type computer), execution will then go to this section of the TSR. The TSR resets the hot key flag and immediately calls the normal BIOS clock procedure. This call is necessary because the clock procedure updates the system clock and controls the timing of many other system operations. When execution returns to the TSR from the BIOS clock procedure, a check is made to see if a hot key was pressed. If not, execution is simply returned to the program that was interrupted by the clock tick.

If a hot key was pressed, this section of the TSR usually has to determine if DOS or BIOS is executing any procedures before transferring execution to the main part of the TSR. The problem here is that for the most part, DOS and BIOS procedures are not reentrant. This means that the system would probably "lock up" if the TSR happened to call a DOS or BIOS procedure that was executing when the clock tick interrupt occurred. We don't have space to show you the details here, but the DOS INT 28H function can be used to determine if it is safe to run a TSR which uses DOS function calls to access disk files, etc.

If a DOS or BIOS function was in process when the clock tick occurred, execution is simply returned to the interrupted program. When the next clock tick occurs, this middle section of the TSR will again check if DOS is available. If no DOS or BIOS functions were executing when the interrupt occurred, execution will go to the main part of the TSR. When the main part of the TSR finishes, execution is returned to the interrupted main program. Note that the hot key flag was previously reset, so that if another clock tick interrupt occurs while the main part of the TSR is executing, the BIOS clock procedure will be executed, but the main part of the TSR will not be called again.

If you want to experiment with TSRs, the Bibliography lists a couple of references which have detailed examples of how to write TSRs.

As you can perhaps see from the preceding discussion, the TSR scheme allows a microcomputer to do limited multitasking, but it is not useful for controlling a multiuser system. In the next section we discuss the scheduling method commonly used in multiuser operating systems.

TIME-SLICE SCHEDULING

In a full-fledged multitasking or multiuser operating system, the part of the operating system which determines when it is time to switch from one task to another is called the *scheduler*, *dispatcher*, or *supervisor*. The most common method of scheduling task switches is the *time-slice* method which we discussed previously. In a simple round-robin implementation of this approach, the CPU executes one task for perhaps 20 ms and then switches to the next task. After all tasks have had their turn, execution returns to the first. In the program in Figure 10-35 we showed you how a programmable timer, priority-interrupt controller, and interrupt-service procedure can be used to implement this type of scheduling. The UNIX operating system and the OS/2 operating system use a more complex time-slice scheduling approach to implement multitasking. The advantage of the time-slice approach in a multiuser system is that all users are serviced at approximately equal time intervals. As more users are added, however, each user gets serviced less often, so each user's program takes longer to execute. This is referred to as *system degradation*. For industrial control operating systems, this variable scheduling is often not appropriate, so a different scheduling method is used.

PREEMPTIVE PRIORITY-BASED SCHEDULING

In a system which uses *preemptive priority-based scheduling*, an executing low-priority task can be interrupted by a higher-priority task. When the high-priority task finishes executing, execution returns to the low-priority task. This approach is well suited to some control applications because it allows the most important tasks to be done first. Priority-interrupt controllers such as the 8259A are often used to set up and manage the task service requests. The Intel RMX 86 operating system uses priority-based scheduling.

Preserving the Environment

The registers, data, pointers, etc., used by an executing task are referred to as its *environment*, *state*, or *context*. When a task switch occurs, the environment of the interrupted task must be saved so that the task can be restarted properly when it receives another time slice. The usual way of preserving the environment is to keep it in a special memory segment or on a stack. Some operating systems keep a separate stack for each task. In either case, when a task switch occurs the operating system saves the environment of the interrupted task and a pointer to the saved environment. When it is time to switch back to that task, the operating system uses the pointer to access the environment it saved. This process is commonly called "context switching."

A less obvious point in a multitasking system is that any global procedures have to be reentrant. This is necessary so that if one task is executing a procedure and its time slice ends, other tasks can use the procedure, and the procedure will still complete correctly when execution returns to the first task. Refer to Figure 5-20 if you need a refresher on reentrancy.

Accessing Resources

Another problem encountered in a multitasking system is assuring that tasks have orderly access to resources such as printers, disk drives, etc. As one example of this, suppose that a user at a terminal needs to read a file from a hard disk and print it on the system printer. Obviously the file cannot be read in from the disk and printed in one of the 20-ms time slices allotted to that user, so several provisions must be made to gain access to the resources and hang on to them long enough to get the job done properly. A flag or *semaphore* in memory is used to indicate whether the disk drive is in use by another task or not. Likewise, another semaphore is used to indicate whether the printer is in use. If a task cannot access a resource because it is busy, the task is said to be *blocked*. Now, rather than making the user type in a print command over and over until the disk drive or the printer is available, most operating systems of this type set up queues of tasks waiting for each resource. When one task finishes with a resource, it resets the semaphore for that resource. The next task in the queue can set the semaphore to indicate the resource is busy and then use the resource.

The Need for Protection

An interesting problem can occur in a multitasking operating system when two or more users attempt to read and change the contents of a memory location at the same time. As an example, suppose that an airline ticket-reservation system is operating on a time-slice basis. Now, further suppose that just before the end of his or her time slice, one user examines the memory location which represents a seat on a plane and finds the seat empty. Another user on the system can then, in his or her time slice, examine the same memory location, find it empty, mark it full, and print out a reservation confirmation on the CRT. When execution returns to the first user, his or her program has already checked the seat during its previous time slice, so it marks the seat full, and prints out a reservation confirmation on the CRT. The two people assigned to the same seat may make nasty remarks about computers unless this problem is solved.

The section of a program where the value of a variable is being examined and changed must be protected from access by other tasks until the operation is complete. The section of code which must be protected is called a *critical region* or *critical section*. A technique called *mutual exclusion* is used to prevent two tasks from accessing a critical region at the same time. In the CHK_N_DISPLAY procedure in Figure 14-27 we showed how a critical region can be protected from an interrupt procedure by simply masking the interrupt. In a time-slice system, however, a semaphore is used to provide mutual exclusion.

Figure 15-3 shows how this can be done with 8086 assembly language instructions. The instruction sequence is the same for each task. If task 1 needs to enter a critical section of code, it first loads the semaphore value for critical-region-busy into AL. The single instruction XCHG AL, SEMAPHORE then swaps the byte in AL with the byte in the memory location named SEMAPHORE. It is important to do this in one instruction so that the time-slice mechanism cannot switch to another task halfway through the exchange and cause our airline problem.

After the semaphore is read in Figure 15-3, it is compared with the busy value. If the critical region is busy, execution will remain in a wait loop for as many time slices as are required for the critical region to become free. If the semaphore value is a 0, indicating not busy, then execution enters the critical region. The XCHG instruction has already set the semaphore to indicate the critical region is busy. After execution of

```
;Instructions for accessing critical region of code protected by semaphone - USER 1
      MOV  AL, 01         ; Load semaphore value for region busy
HOLD: XCHG AL, SEMAPHORE  ; Swap and set semaphore
      CMP  AL, 01         ; Check if region is busy
      JE   HOLD           ; Yes, loop until not busy. No enter critical region of code.
;     Instructions which access critical region are inserted here
      MOV  SEMAPHORE, 00  ; Reset semaphore to make critial region available to others.


;Instructions for accessing critical region of code protected by semaphone - USER 2
      MOV  AL, 01         ; Load semaphore value for region busy
HOLD: XCHG AL, SEMAPHORE  ; Swap and set semaphore
      CMP  AL, 01         ; Check if region is busy
      JE   HOLD           ; Yes, loop until not busy. No enter critical region of code.
;     Instructions which access critical region are inserted here
      MOV  SEMAPHORE, 00  ; Reset semaphore to make critial region available to others.
```

FIGURE 15-3 8086 assembly language sequences showing how a flag or semaphore can be used to provide mutual exclusion for a critical region of code.

the critical region finishes, the MOV SEMAPHORE, 00 instruction resets the semaphore to indicate that the critical region is no longer busy. Task 2 can then swap the semaphore and access the critical region when needed. The semaphore functions in the same way as the "occupied" sign on a restroom of a plane or train. If you mentally try interrupting each sequence of instructions at different points, you should see that there is no condition where both tasks can get into the critical region at the same time.

Another region that requires protection is the operating system code. Most single-user operating systems such as DOS do little to prevent user programs from corrupting the operating system code and data areas. The usual results of this and Murphy's law are that an incorrect address in a user program may cause it to write over critical sections of the operating system. The system then "locks up" and the only way to get control again is to reboot the system. In a multitasking system this is intolerable, so several methods are used to protect the operating system.

The major method is to construct the operating system in two or more *layers*. Figure 15-4 shows an "onionskin" diagram for a two-layer operating system. The basic principle here is that the inner circle represents the code and data areas used by the operating system. The outer layer represents the code and data areas of user programs or tasks that are being run under control of the operating system. The inner layer is protected because user programs can only access operating system resources through very specific mechanisms rather than a simple, accidental call or jump. Devices in the Motorola MC68000 family of microprocessors, for example, are designed to accommodate a two-level structure such as this. The MC68000 family devices have two modes of operation, user and supervisory. Certain privileged instructions which affect the operating system can only be executed when the processor is in supervisory mode. As we discuss in great detail later, the Intel 80286, 80386, and 80486 microprocessors have hardware features which allow up to four levels of protection to be built into a system. The 80286, 80386, and 80486 microprocessors also provide a hardware mechanism which can be used to protect tasks from each other.

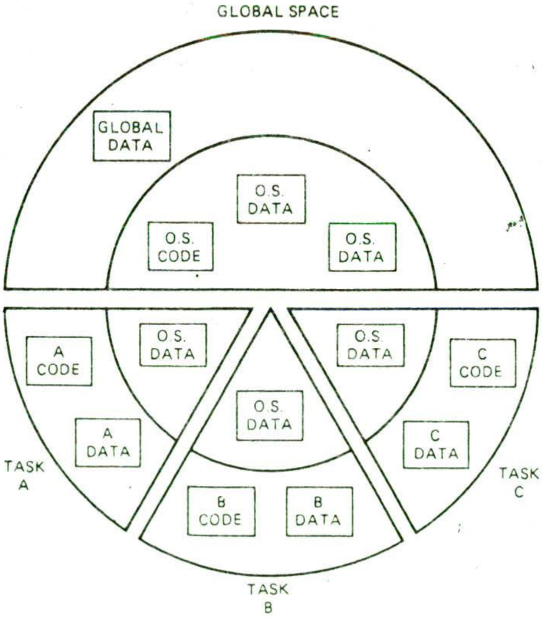


FIGURE 15-4 "Onionskin" diagram showing two-level-protection scheme for multitasking operating system. (*Intel Corporation*)

Memory Management

INTRODUCTION

There are two major reasons why memory must be specifically managed in a multitasking operating system. The first reason is that the physical memory is usually not large enough to hold the operating system and all of the application programs that are being executed by the different users. The second reason is to make sure that executing tasks do not access protected areas of memory. Some memory management can be done by the operating system software, but complete memory management and protection require the aid of hardware called a *memory-management unit* or MMU. Before we get into the operation of an MMU, we want to give you a little background on other methods used to solve the limited memory problem.

OVERLAYS

A common problem, even in older, single-user systems, is that the physical memory is not large enough to hold, for example, an assembler and the program being assembled. The traditional solution to this problem is to write the assembler in modules and use an *overlay* scheme. When the assembler is invoked, the executive module of the assembler is loaded into memory, and an additional block of memory space called the *overlay area* is reserved for the assembler. The first module of the assembler is loaded into this overlay area. When the assembler reaches a point where it needs the next module, it reads that module, referred to as an *overlay*, from disk into the overlay area reserved in memory. When the assembler reaches a point where it needs another overlay, it reads that overlay from disk and loads it into the same overlay area in memory. The overlay approach is commonly used with assemblers, compilers, word processors, and spreadsheet programs. Incidentally, the Borland Turbo C + + tools we introduced you to in Chapter 12 can be used to develop an overlay type program.

BANK SWITCHING, EXPANDED MEMORY, AND EXTENDED MEMORY

Another approach traditionally used to expand the available memory in a microcomputer is *bank switching*. Early microprocessors such as the Intel 8085 have only 16 address lines, so they can directly address only 64 Kbytes of memory. Figure 15-5 shows how the amount

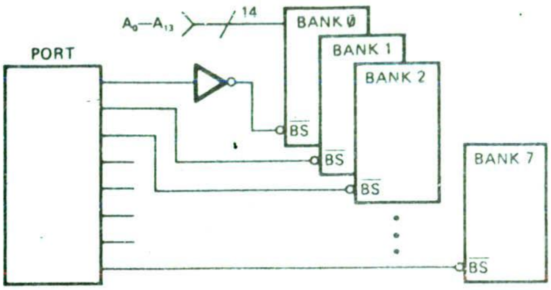


FIGURE 15-5 Block diagram showing how microcomputer memory can be expanded with bank switching.

of memory accessible in a system such as this can be expanded beyond the address limit. The hardware is configured so that when the power is first turned on, the 16-Kbyte bank labeled bank 0 is enabled. Let's assume that this bank occupies system address space 4000H–7FFFH and that system address lines A0–A13 are used to address the bytes in this bank.

To switch to bank 1, a byte which turns off bank 0 and turns on bank 1 is output to the selection port. The bank 1 devices now occupy the address space 4000H–7FFFH and system address lines A0–A13 are used to address the bytes in this bank. Any of the other banks can be switched into the 4000H–7FFFH memory window by simply sending the appropriate word to the control port. As you can see; this bank-switching scheme allows the processor to access 8 banks of 16 Kbytes each or a total of 128 Kbytes through a 16-Kbyte window in the processor address space. Let's see how this scheme is used in IBM PC- and PS/2-type microcomputers.

The 8086 or 8088 processor used in PC-type microcomputers can address up to 1 Mbyte of memory. At the time the IBM PC was developed, it seemed inconceivable that anyone would ever need more than 640 Kbytes of memory for application programs, so all the address space above 640 Kbytes was reserved for the system BIOS, the video frame buffer, and system uses, as shown in Figure 15-6. Also, since the processor could address only 1 Mbyte of memory, DOS was designed to directly address only 1 Mbyte.

As the memory needs of application programs such as spreadsheets and databases banged into the 640-Kbyte limit, designers again looked to bank switching as a means to overcome this limit. The result was the Lotus-Intel-Microsoft Expanded Memory Standard, LIM/EMS 3.2. This combination hardware-software standard has been widely implemented.

The hardware for this expanded memory is often implemented as a plug-in board which contains up to 8 Mbytes of 16-Kbyte pages (banks) and bank-switch registers. The bank-switch registers are used to control which pages from the expanded memory are selected. As shown along the left side of Figure 15-6, in a LIM/EMS 3.2 system the four 16-Kbyte pages selected from the expanded memory at a particular time are mapped into the system address space between C800H and D7FFH. This address space was chosen for the expanded memory window because it is not usually used for system functions. The newer LIM/EMS 4.0 standard allows 16-Kbyte pages to be mapped into any system address space that is not populated with ROM or RAM, and it allows the expanded memory to contain up to 32 Mbytes. As shown along the right side of Figure 15-6, LIM/EMS 4.0 allows pages above the 640-Kbyte boundary and additional pages in the 384-Kbyte region below the boundary.

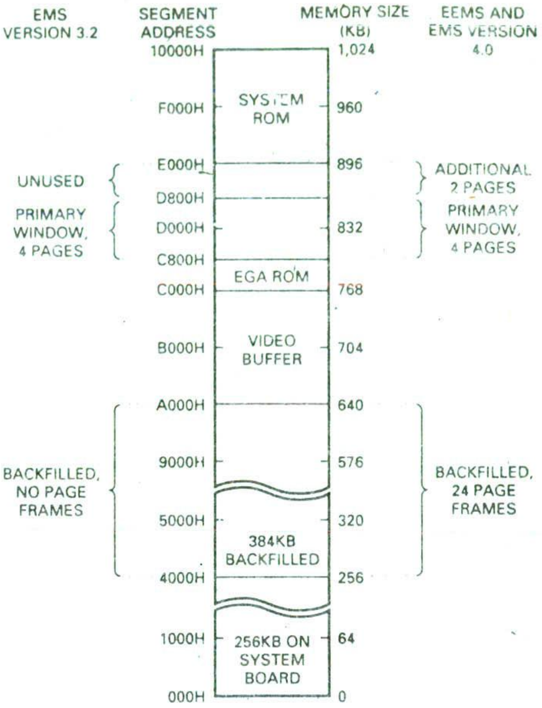


FIGURE 15-6 Memory maps for LIM/EMS 3.2 and LIM/EMS 4.0 expanded memory standards.

The software part of either EMS standard includes a driver program called EMM.SYS. This driver program is installed in memory by including the statement device = emm.sys in the CONFIG.SYS program which runs when you boot your system. The EMM.SYS driver contains the functions which allow application programs to allocate and access expanded memory. The expanded memory functions are called with a software INT 67H. The value in AH determines the specific function that is called. The complete list of expanded memory functions is extensive, but to give you an idea of some of what is available, Figure 15-7 shows a few of the functions. Basically, an application program must use these functions to allocate enough expanded memory for its code and data, switch in pages as needed, and deallocate the expanded memory when it terminates so that the memory is available for the next program. Incidentally, MS DOS versions 4.0 and later support LIM/EMS 4.0.

As we discuss in detail later, the 80286, 80386, and 80486 microprocessors have more address lines than

EXPANDED MEMORY FUNCTION	CALL WITH	RETURNS
GET STATUS	AH = 40H	AH = STATUS

GET PAGE FRAME ADDRESS	AH = 41H	AH = STATUS
		BX = PAGE FRAME SEGMENT

GET NUMBER OF EXPANDED MEMORY PAGES	AH = 42H	AH = STATUS
		BX = AVAILABLE PAGES
		DX = TOTAL PAGES

ALLOCATE EXPANDED MEMORY PAGES	AH = 43H
	BX = NO. OF PAGES
		AH = STATUS
		DX = EMM HANDLE

MAP EXPANDED MEMORY PAGE	AH = 44H
	AL = PHYSICAL PAGE
	BX = LOGICAL PAGE
	DX = EMM HANDLE
		AH = STATUS

RELEASE EXPANDED MEMORY PAGES	AH = 45H
	DX = EMM HANDLE
		AH = STATUS

GET EMM VERSION	AH = 46H	AH = STATUS
		AL = VERSION

FIGURE 15-7 Examples of EMM functions called through INT 67H.

an 8086 and can directly address considerably more memory. Memory located in the address space above 1 Mbyte is commonly referred to as *extended memory* or *XMS memory*. If a system using one of these processors is running under a version of DOS before 5.0, however, it still has the 1-Mbyte memory limit imposed by DOS. In other words, the extended memory in a system is invisible to DOS and will not be used for programs. There are three common cures for this problem.

One solution is to use a memory-management-device driver program which allows the extended memory to function as expanded memory. Another solution is to use a "DOS extender" program such as Phar Lap Software's 386/DOS extender or A.I. Architect's OS/x86. These programs operate under DOS, so they use the familar DOS commands, but they allow programs to take advantage of the advanced features of the 80286, 80386, and 80486 processors. The third solution to the DOS memory limit is to switch to an operating system such as Microsoft's OS/2, which is designed to take advantage of the addressing range and other features of the newer processors.

The expanded memory scheme we described in the preceding section makes more memory available to a program, but it has several disadvantages. One disadvantage is that the system must contain enough expanded memory for the largest program to be run. With today's large programs this could be a major expense. A second disadvantage of expanded memory is that application programs must manage the switching of pages in and out of the expanded memory window. This adds overhead to the execution time, and if a program is modified, the switching points may have to be changed. Still another disadvantage is that operating system and user-task protection are not easily implemented. The virtual memory scheme we discuss next helps solve these problems.

VIRTUAL MEMORY AND MMUs

Virtual memory is basically an extension of the memory caching scheme we discussed in Chapter 11. To refresh your memory of a cache system, take another look at Figure 11-11. The virtual memory scheme simply adds a hard-disk drive to the memory hierarchy. The hard-disk drive becomes the main program and data memory, the DRAM functions as an intermediate cache, and the SRAM cache functions as a high-speed cache for the DRAM. In a virtual memory system the code and data segments currently being used for program execution are loaded from the disk into DRAM and accessed by the cache controller as needed. If an executing program needs a segment that is not currently in DRAM, the required segment is read in from the disk to the DRAM main memory. If the DRAM is full, one of the segments in the DRAM is swapped out to the disk to make room, and the required segment is swapped into DRAM.

There are three different ways of setting up the code and data blocks to be swapped in and out of DRAM. One scheme is to swap segments. The advantage of segment swapping is that segments correspond to the code and data structures in the program. The disadvantage of the segment scheme is that with processors such as the 80386 and 80486, segments can be very large. The time required to swap in a large segment would appreciably slow down the execution of a program. Also, it is often hard to fit variable-sized segments in memory. A second swapping scheme uses fixed-length pages of typically 4 Kbytes each. These small pages can be quickly swapped in and out of memory, but they don't correspond to the logical structure of the program. A third approach, implemented in the 80386 and 80486 microprocessors, allows a programmer to write a program using logical segments and divide the segments into 4-Kbyte pages for swapping in and out of physical memory.

The term virtual here refers to memory space that appears to be present from a programmer's viewpoint but is not physically present in the DRAM main memory. In other words, if you are writing a program for a system with virtual memory, you can create segments as if you had, for example, a gigabyte of memory space, even though the system has only perhaps 4 Mbytes of physical memory. The virtual memory space can be much larger than the physical memory, because all of the logical segments are not present in physical memory at any one time. As with the SRAM cache scheme, a virtual memory system works because most programs only need small sections of code and data at a particular time.

Virtual memory can be managed totally by the operating system, but most microcomputer systems use a hardware device called a *memory-management unit* or MMU to assist in the process. The Intel 80286, 80386, and 80486 and the Motorola MC68030 and MC68040 have a complete MMU integrated on the chip with the CPU. Separate MMUs are available for use with other processors. In either case the MMU is functionally positioned between the processor and the actual memory. Figure 15-8, page 542, shows an overview of how the MMUs in the 286, 386, and 486 processors manage segment-based virtual memory. The first step in explaining this is to clarify the terms logical address and physical address.

When you write an assembly language program, you usually refer to addresses by name. The addresses you work with in a program are called *logical addresses*, because they indicate the logical positions of code and data. An example of this is the 8086 instruction JNZ

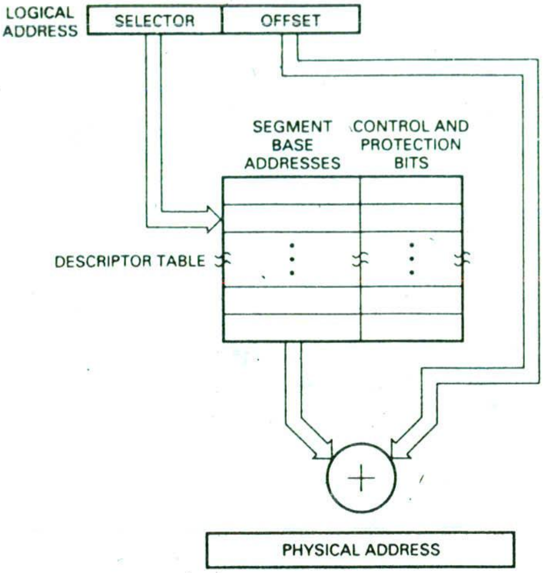


FIGURE 15-8 Block diagram showing how segment-based virtual memory is implemented in 80286, 80386, and 80486 processors.

NEXT. The label NEXT represents a logical address that execution will go to if the zero flag is not set. When an 8086 program is assembled, each logical address is represented with a 16-bit offset and a 16-bit segment base. The 8086 BIU then produces the actual physical memory address by simply adding these two parts together, as explained many times previously.

When a program is assembled or compiled to run on a system with an MMU, each logical or virtual address is also represented by two components, but the components function differently. In a segment-oriented system such as an 80286, the upper 16-bit component is referred to as a *segment selector*, and the lower component is referred to as the *offset*. As shown in Figure 15-8, the MMU uses the segment selector to access a *descriptor* for the desired segment in a table of descriptors in memory. A descriptor is a series of memory locations that contain the physical base address for a segment, the privilege level of the segment, and some control bits.

The selectors for the 80286, 80386, and 80486 have 14 address bits and 2 privilege-level bits. The 14 address bits in the selector can select any one of 16,384 descriptors in the descriptor table. Since each descriptor represents a segment, this means that a program can access up to 16,384 segments. For an 80286 the offset part of the virtual address is 16 bits, so each segment can contain up to 64 Kbytes. The logical or virtual address space accessible by an 80286 then is 16,384 segments × 65,536 bytes/segment, or about 1 Gbyte. What this means is that the operating system and other programs can function as if a gigabyte of memory were available.

The physical memory is the amount of RAM and ROM actually present in the system. For this example let's assume that the MMU has 24 address lines so it can address 16 Mbytes of physical memory. Remember from our previous discussion that the physical memory, whatever its actual size, is simply a holding place for the segments currently being used by the operating system and user programs.

When the MMU receives a logical address from the CPU, it checks to see if that segment is currently in the physical memory. If the segment is present in physical memory, the MMU adds the offset component of the address to the segment base component of the address from the segment descriptor to form the physical address. It then outputs the physical address to memory on the memory address bus. The addressed code or data word is returned to the CPU on the data bus.

If the MMU finds that the segment specified by the selector part of the logical address is not in memory, it sends an interrupt signal to the CPU. In response to the interrupt, the operating system executes an interrupt procedure which reads the desired code or data segment from disk and loads it into the physical memory. The MMU then computes and outputs the physical address as described before. The operation is semiautomatic, so other than a slight delay, the user is not aware that the segment had to be loaded. In a well-structured system with a reasonably large amount of physical memory, the *hit rate* may be 90 to 95 percent.

When the CPU or smart MMU wants to load a segment from disk into physical memory, it must first make space for it in the physical memory. Depending on the system, it may do this by compacting the segments already present and changing the descriptors to point to the new physical locations or by swapping the segment being brought in with one currently in physical memory. To help in deciding which segment to swap back to memory, many systems use an *accessed* bit in the descriptor to keep track of how many times the segment has been used. A low-use segment is the most likely candidate to swap back to disk. Some virtual memory systems also have a *dirty* bit in each descriptor. This bit will be set if the contents of a segment have been changed. If the dirty bit is set, a segment must be written back to disk when its space is needed. If the dirty bit is not set, then the segment has not been altered, and the copy of the segment on disk is current. In this case the segment can just be overwritten by the new segment. This check saves the time that would be required to write the segment to disk.

The use of a descriptor table to translate logical addresses to physical addresses has another major advantage besides making virtual memory possible. The selector component of each logical address contains 2 bits which represent the privilege level of the program section requesting access to a segment. The descriptor for each segment contains 2 bits which represent the privilege level of that segment. When an executing program attempts to access a segment, the MMU compares the privilege level in the selector with the privilege level in the descriptor. If the segment selector has the same or a greater privilege level, then the MMU allows the segment to be accessed. If the selector privilege level is lower than the privilege level in the descriptor, the MMU refuses the access and sends an interrupt signal to the CPU indicating a privilege-level violation. As you can see, privilege bits and this indirect method of producing physical addresses provides a mechanism for protecting segments such as those containing the operating system kernel from application programs.

To summarize then, an MMU is used to manage virtual memory. The MMU uses a descriptor table to translate logical or virtual program addresses to physical addresses. This indirect approach makes possible a virtual address space much larger than the physical address space. The indirect approach also makes it possible to protect a memory segment from access by a program section with a lower privilege level. You will meet all these concepts again in the following sections, where we discuss the 80286, 80386, and 80486 microprocessors which have integrated MMUs.

THE INTEL 80286 MICROPROCESSOR

Introduction

The needs of a multitasking/multiuser operating system include environment preservation during task switches, operating system and user protection, and virtual memory management. The Intel 80286 was the first 8086 family processor designed to make implementation of these features relatively easy. The 80286 was used as the CPU in the IBM PC/AT and its clones, in the IBM PS/2 Model 50, and in the IBM PS/1. Although the 80286 has to a large extent been superseded by the 80386, the 80386SX, and the 80486, there are still many 80286-based systems in use and more 80286 systems being sold. Therefore, we will use a little space to tell you about the basic operation of an 80286.

80286 Architecture, Signals, and System Connections

As you can see in the block diagram in Figure 15-9, an 80286 contains four separate processing units.

The *bus unit* (BU) in the device performs all memory and I/O reads and writes, prefetches instruction bytes, and controls transfer of data to and from processor extension devices such as the 80287 math coprocessor.

The *instruction unit* (IU) fully decodes up to three prefetched instructions and holds them in a queue, where the execution unit can access them. This is a further example of how modern processors keep several instructions "in the pipeline" instead of waiting to finish one instruction before fetching the next.

The *execution unit* (EU) uses its 16-bit ALU to execute instructions it receives from the instruction unit. When operating in its real address mode, the 80286 register set is the same as that of an 8086 except for the addition of a 16-bit machine status word (MSW) register, which we will discuss later.

The *address unit* (AU) computes the physical addresses that will be sent out to memory or I/O by the BU. The 80286 can operate in one of two memory address modes, *real address mode* or *protected virtual address mode*. If the 80286 is operating in the real address mode, the address unit computes addresses using a segment base and an offset just as the 8086 does. The familiar CS, DS, SS, and ES registers are used to hold the base addresses for the segments currently in use. The maximum physical address space in this mode is 1 Mbyte, just as it is for the 8086.

If an 80286 is operating in its *protected virtual address mode* (protected mode), the address unit functions as a complete MMU. In this address mode the 80286 uses all 24 address lines to access up to 16 Mbytes of physical memory. In protected mode it also provides up to a gigabyte of virtual memory using the descriptor table scheme shown in Figure 15-8.

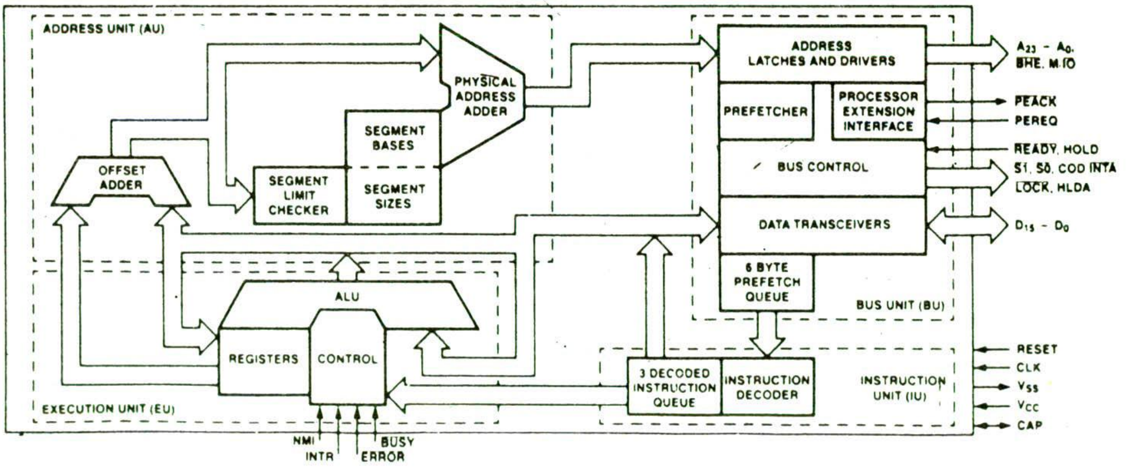


FIGURE 15-9 80286 internal block diagram. (*Intel Corporation*)

Figure 15-10 shows the 68-pin package that is usually used for an 80286, and Figure 15-11 shows how an 8086 is connected with some other components to form a simple system. Keep thse figures handy as we work our way around the major pins of the 80286. Many of the signals of the 80286 should be familiar to you from our discussion of the 8086 signals in Chapter 7.

The 80286 has a 16-bit data bus and a 24-bit nonmultiplexed address bus. The 24-bit address bus allows the processor to access 16 Mbytes of physical memory when operating in protected mode. Memory hardware for the 80286 is set up as an odd bank and an even bank, just as it is for the 8086. The even bank will be enabled when A0 is low, and the odd bank will be enabled when $\overline{\text{BHE}}$ is low. To access an aligned word, both A0 and $\overline{\text{BHE}}$ will be low. External buffers are used on both the address and the data bus.

From a control standpoint, the 80286 functions similarly to an 8086 operating in maximum mode. Status signals $\overline{\text{S0}}$, $\overline{\text{S1}}$, and M/$\overline{\text{IO}}$ are decoded by an external 82288 bus controller to produce the control bus, read, write, and interrupt-acknowledge signals.

The HOLD, HLDA, INTR, $\overline{\text{INTA}}$, (NMI), $\overline{\text{READY}}$, and $\overline{\text{LOCK}}$ and RESET pins function basically the same as they do on an 8086. An external 82284 clock generator is used to produce a clock signal for the 80286 and to synchronize RESET and $\overline{\text{READY}}$ signals.

The final four signal pins we need to discuss here are used to interface with processor extensions (coprocessors) such as the 80287 math coprocessor. The *processor extension request* (PEREQ) input pin will be asserted by a coprocessor to tell the 80286 to perform a data transfer to or from memory for it. When the 80286 gets around to do the transfer, it asserts the *processor extension acknowledge* ($\overline{\text{PEACK}}$) signal to the coprocessor to let it know the data transfer has started. Data transfers are done through the 80286 in this way so that the coprocessor uses the protection and virtual memory capability of the MMU in the 80286. The $\overline{\text{BUSY}}$ signal input on the 80286 functions the same as the $\overline{\text{TEST}}$ input does on the 8086. When the 80286 executes a WAIT instruction, it will remain in a WAIT loop until it finds the $\overline{\text{BUSY}}$ signal from the coprocessor high. If a coprocessor finds some error during processing, it will assert the $\overline{\text{ERROR}}$ input of the 80286. This will cause the 80286 to automatically do a type 16H interrupt call. An interrupt-service procedure can be written to make the desired response to the error condition.

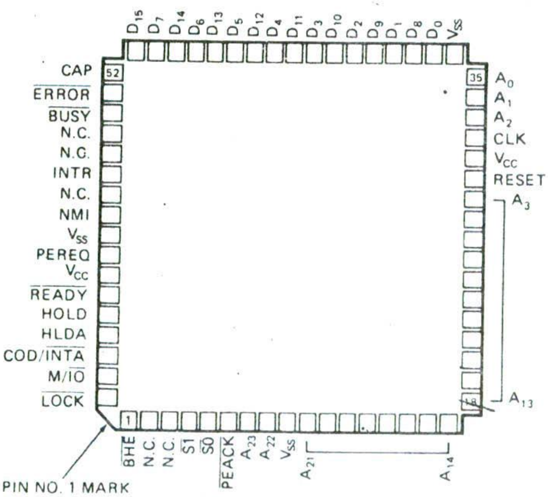


FIGURE 15-10 Pin diagram for 80286 microprocessor. (*Intel Corporation*)

The machine cycle waveforms for the 80286 are very similar to those of the 8086 that we showed and discussed in earlier chapters. You should be able to work your way through them in the Intel 80286 data sheets if you need that type of information.

As we mentioned before, the 80286 is used as the CPU in the IBM PC/AT and its clones. These AT-type machines use the AT/ISA bus shown in Figure 11-7*b* to interface with a CRT controller card, disk controller cards, and other peripheral cards.

80286 Real Address Mode Operation

After the 80286 is reset, it starts executing in its real address mode. This mode is referred to as real because physical memory addresses are produced by directly adding an offset to a segment base, just as they are in an 8086. In this mode the 80286 can address up to 1 Mbyte of physical memory and functions essentially as a "souped-up" 8086. Due to the extensive pipelining and other hardware improvements, the 80286 will execute most programs several times faster than an 8086 with the same-frequency clock signal.

When operating in real address mode, the interrupt-vector table of the 80286 is located in the first 1 Kbyte of memory, just as it is for an 8086, and the response to an interrupt is the same as that of an 8086. As shown in Figure 15-12 the 80286 has several additional built-in interrupt types. Some of these types will not make much sense until we dig a little deeper into the operations of the 80286 and the 80386, but while we are here we will introduce you to a few new terms used in Figure 15-12.

The 80186 and later processors separate interrupts into two categories, interrupts and exceptions. Asynchronous external events which affect the processor through the INTR or NMI input are referred to as interrupts. An exception-type interrupt is generated by some error condition that occurred during the execution of an instruction. Dividing by zero is an example of an operation that will cause an exception. Software interrupts produced by the INT n instruction are classified as exceptions, because they are synchronous with the processor.

Exceptions are further divided into faults and traps. Faults are exceptions that are detected and signaled before the faulting instruction is executed. The segment-not-present exception is an example of a fault. Traps are exceptions which are reported after the instruction which caused the exception executes. The divide-by-zero exception and the INT n interrupts are examples of traps.

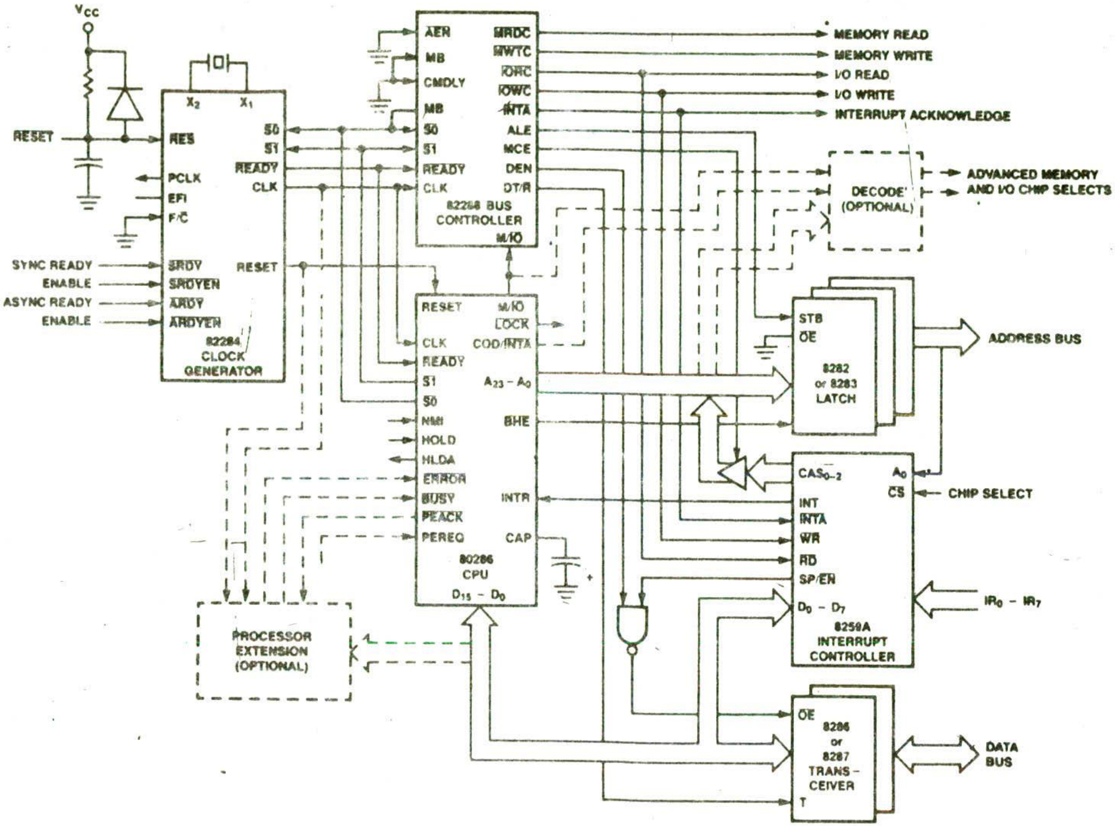


FIGURE 15-11 Circuit connections for simple 80286 system. (*Intel Corporation*)

FUNCTION	INTERRUPT NUMBER
DIVIDE ERROR EXCEPTION	0
SINGLE STEP INTERRUPT	1
NMI INTERRUPT	2
BREAKPOINT INTERRUPT	3
INTO DETECTED OVERFLOW EXCEPTION	4
BOUND RANGE EXCEEDED EXCEPTION	5
INVALID OPCODE EXCEPTION	6
PROCESSOR EXTENSION NOT AVAILABLE EXCEPTION	7
INTERRUPT TABLE LIMIT TOO SMALL	8
PROCESSOR EXTENSION SEGMENT OVERRUN INTERRUPT	9
INVALID TASK STATE SEGMENT	10
SEGMENT NOT PRESENT	11
STACK SEGMENT OVERRUN OR NOT PRESENT	12
SEGMENT OVERRUN EXCEPTION	13
RESERVED	14 15
PROCESSOR EXTENSION ERROR INTERRUPT	16
RESERVED	17 31
USER DEFINED	32-255

FIGURE 15-12 80286 interrupt types. (*Intel Corporation*)

80286 Protected-Mode Operation

As we said before, after a reset the 80286 operates in real address mode. On an 80286-based system running under MS DOS or a similar operating system, the 80286 is left in real address mode because current versions of DOS are not designed to take advantage of the protected-mode features of the 80286. If an 80286-based system is running an operating system such as Microsoft's OS/2, which uses the protected mode, the real mode will be used to initialize perhipheral devices, load the main part of the operating system from disk into memory, load some registers, enable interrupts, set up descriptor tables, and switch the processor to protected mode. The first step in switching an 80286 to protected mode is to set the protection enable bit in the *machine status word* (MSW) register in the 80286. Figure 15-13*a*, page 546, shows the format for the MSW. Bits 1, 2, and 3 of the MSW are for the most part used to indicate whether a processor extension (coprocessor) is present in the system or not. Bit 0 of the MSW is used to switch the 80286 into protected mode. To change bits in the MSW you load the desired word in a register or memory location and execute the *load machine status word* (LMSW) instruction. The final step to get the 80286 operating

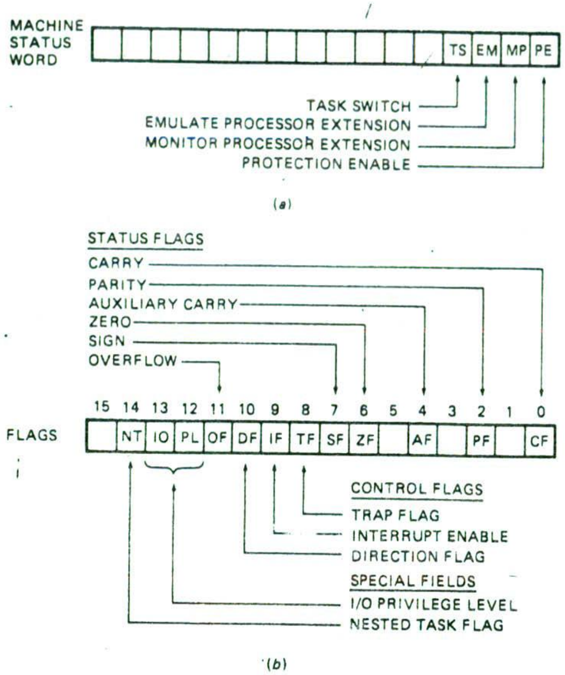


FIGURE 15-13 (*a*) 80286 machine status word bits. (*b*) 80286 flag register bits. (*Intel Corporation*)

in protected mode is to execute an intersegment jump to the start of the main system program. This jump is necessary to flush the instruction byte queue because in protected mode the queue functions differently from the way it does in real mode.

Switching an 80286 to protected mode enables the integrated MMU to provide virtual memory and protection. As we described in an earlier section on virtual memory, a 286 virtual address consists of a 16-bit selector and a 16-bit offset. The MMU uses 14 bits of the selector to access a descriptor for the desired segment in a table of descriptors. The descriptor contains the 24-bit physical base address, the privilege level, and some control bits for the segment. If the privilege level contained in the selector is as high as or higher than the privilege level contained in the descriptor, then access to the segment will be allowed. If not, an exception will be generated. The MMU also checks the "P" bit in the descriptor to determine if the segment is present in physical memory. If not, the MMU will generate a segment-not-present exception. The service procedure for this exception will load the segment in memory and return to the interrupted program. If the memory access meets the privilege level test and the segment is present in physical memory, the MMU will add the 16-bit offset from the logical address to the 24-bit base address from the descriptor to produce the 24-bit physical address for the desired byte or word in the segment. Remember that in protected mode an 80286 uses all 24 address lines, so it can address 16 Mbytes of memory instead of just the 1 Mbyte addressable in real mode.

Once an 80286 is switched into protected mode by executing the LMSW instruction, the only way to get an 80286 back to its real address mode is by resetting the system. The 80286 was designed this way so that a "clever" programmer could not switch the system back into real address mode to defeat the protection schemes in protected mode. Unfortunately, this design also prevents an operating system running in protected mode on an 80286 from easily switching back to real mode to run a section of an 8086 real-mode program during a time slice. In other words, an 80286 operating in protected mode cannot easily multitask a mixture of programs with 8086 segment-offset-type addressing and 80286 selector-offset-type addressing. For this and other reasons, relatively little software has been written to take advantage of the memory-management and protection features available in the 80286 protected mode. The designs of the 80386 and 80486 processors solved the 80286 problems and added other features which make multitasking easier to implement. Much of the new software written during the lifetime of this book will utilize the advanced features of the 386 and 486. Therefore, we decided that the limited space we have available is better used to discuss the details of how the 386 and 486 manage virtual memory and provide protection. The protected mode operation of the 386 is very similar to that of the 80286, so if you have to work on a protected-mode 80286 system, you should have little difficulty "going back."

80286 New and Enhanced Instructions

From a software standpoint the 80286 was designed to be upward-compatible from the 8086 so that the huge amount of software developed for the 8086/8088 could be easily transported to the 80286. The instruction set of the 80286 and later processors are "supersets" of the 8086 instructions. Here's a brief description of the new and enhanced instructions available on the 80286.

Real- or protected-mode instructions

INS—Input string.

OUTS—Output string.

PUSHA—Push eight general-purpose registers on stack.

POPA—Pop eight general-purpose registers from stack.

PUSH immediate—Push immediate number on stack.

SHIFT/ROTATE destination, immediate—Shift or rotate destination register or memory location specified number of bit positions.

IMUL destination, immediate—Signed multiply destination by immediate number.

IMUL destination, multiplicand, immediate multiplier—Signed multiply, result in specified destination.

ENTER—Set up stack frame in procedure. Saves BP, points BP to TOS, and allocates stack space for local variables.

LEAVE—Undo ENTER actions before RET in procedure.

BOUND—Causes a type 5 execution if value in specified register is not within the specified range for an array.

LMSW—Load machine status word (LMSW) is used to switch the 80286 from real mode to protected mode.

Protected-mode instructions

> NOTE: We postponed much of the discussion of protected mode to a later section on the 386 processor, so many of these instructions will be much more understandable to you after you read that section.

CTS—Clear task-switched flag in machine status word.

LGDT—Load global descriptor table register from memory.

SGDT—Store global descriptor table register contents in memory.

LIDT—Load interrupt descriptor table register from memory.

LLDT—Load selector and associated descriptor into LDTR.

SLDT—Store selector from LDTR in specified register or memory.

LTR—Load task register with selector and descriptor for TSS.

STR—Store selector from task register in register or memory.

LMSW—Load machine status register from register or memory.

SMSW—Store machine status word in register or memory.

LAR—Load access rights byte of descriptor into register or memory.

LSL—Load segment limit from descriptor into register or memory.

ARPL—Adjust requested privilege level of selector (down only).

VERR—Determine if segment pointed to by selector is readable.

VERW—Determine if segment pointed to by selector is writeable.

THE INTEL 80386 32-BIT MICROPROCESSOR

Introduction

Some of the limitations of the 80286 microprocessor are that it has only a 16-bit ALU, its maximum segment size is 64 Kbytes, and it cannot easily be switched back and forth between real and protected modes. The Intel 80386 microprocessor was designed to overcome these limits, while maintaining software compatibility with the 80286 and earlier processors. The 80386 has a 32-bit ALU, so it can operate directly on 32-bit data words. 80386 segments can be as large as 4 Gbytes and a program can have as many as 16,384 segments. The virtual address space then is 16,384 segments × 4 Gbytes, or about 64 Tbytes (terabytes). A 32-bit address bus allows an 80386 to address up to 4 Gbytes of physical memory. The 80386 has a "virtual 8086" mode, which allows it to easily switch back and forth between 80386 protected-mode tasks and 8086 real-mode tasks. Later we will discuss 80386 memory addressing, protection, and operating modes, but for now we want to discuss the hardware operation and system connections.

80386 Architecture, Pins, and Signals

The 80386 processor is available in two different versions, the 386DX and the 386SX. The 386DX has a 32-bit address bus and a 32-bit data bus. It is packaged in the 132-pin ceramic pin grid array package shown in Figure 15-14*a*, page 548. The 386SX, which is packaged in the 100-pin flatpack shown in Figure 15-14*b*, has the same internal architecture as the 386DX, but it has only a 24-bit address bus and a 16-bit data bus. The lower cost package and the ease of interfacing to 8-bit and 16-bit memory and peripherals make the 386SX suitable for use in lower cost systems. The trade-off here, of course, is that the 386SX address range and memory transfer rate are lower than those of the 386DX. Any reference to the 386 in the rest of this chapter will mean the 386DX unless specifically indicated otherwise.

Figure 15-15, page 548, shows the major signal groups for a 386DX. Most of these signals should be familiar to you from the discussions of earlier processors. Let's work our way around the device to pick up the new ones.

The clock signal applied to the 386 CLK2 input is internally divided by 2 to produce the clock signal which actually drives processor operations. For 33-MHz operation then, a 66-MHz signal is applied to the CLK2 input by an external clock generator such as the 82384.

The 386 address bus consists of the A2–A31 address lines and the byte enable lines BE0#–BE3#. The BE0#–BE3# lines are decoded from internal address signals A0 and A1 and function very similarly to the way A0 and BHE function in an 8086 or 80286 system. The 386 has a 32-bit data bus, so memory can be set up as four byte-wide banks. The BE0#–BE3# signals function as enables for the four banks. These individual enables allow the 386 to transfer bytes, words, or double words to and from memory. Incidentally, the # symbol after the BE signal names indicates that these signals are active low.

The bus cycle definition signals identify the type of operation that is occurring during a bus cycle. The WR/R# signal indicates whether a read or write operation is taking place and the D/C# indicates whether the bus operation is a data read/write or a control-word transfer such as an op-code fetch. M/IO# indicates whether the operation is a memory or a direct input/output operation. Incidentally, the 386 direct I/O port structure

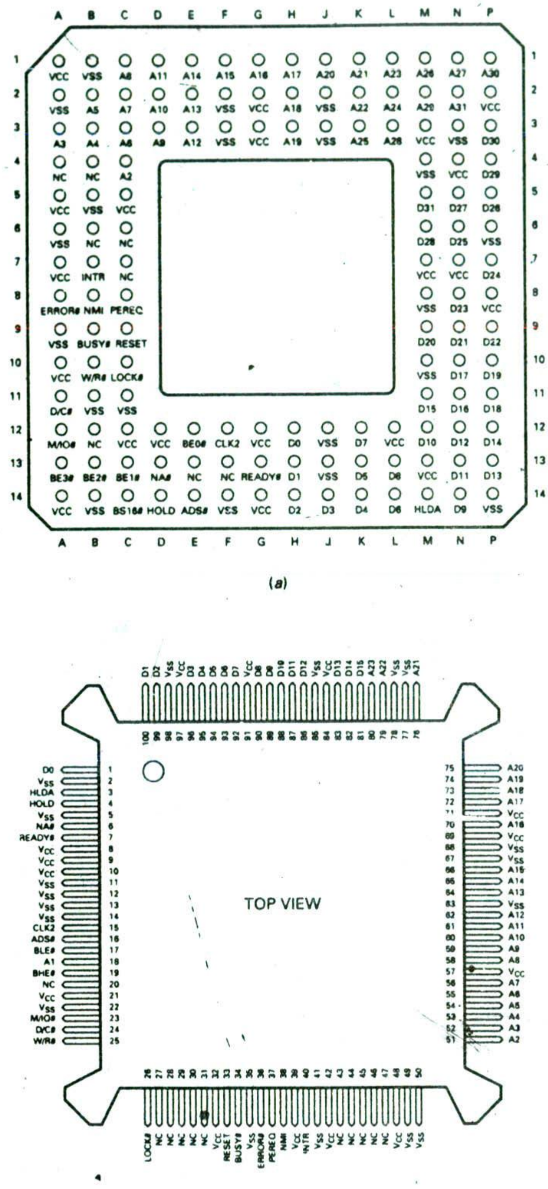

A B C D E F G H J K L M N P
1 VCC VSS A8 A11 A14 A15 A16 A17 A20 A21 A23 A26 A27 A30
2 VSS A5 A7 A10 A13 VSS VCC A18 VSS A22 A24 A29 A31 VCC
3 A3 A4 A6 A9 A12 VSS VCC A19 VSS A25 A28 VCC VSS D30
4 NC NC A2 VSS VCC D29
5 VCC VSS VCC D31 D27 D26
6 VSS NC NC D28 D25 VSS
7 VCC INTR NC VCC VCC D24
8 ERROR# NMI PEREC VSS D23 VCC
9 VSS BUSY# RESET D20 D21 D22
10 VCC W/R# LOCK# VSS D17 D19
11 D/C# VSS VSS D15 D16 D18
12 M/IO# NC VCC VCC BE0# CLK2 VCC D0 VSS D7 VCC D10 D12 D14
13 BE3# BE2# BE1# NA# NC NC READY# D1 VSS D5 D8 VCC D11 D13
14 VCC VSS BS16# HOLD ADS# VSS VCC D2 D3 D4 D6 HLDA D9 VSS
(a)
D1 D2 Vss Vcc D3 D4 D5 D6 D7 Vcc D8 D9 D10 D11 D12 Vss Vcc D13 D14 D15 A23 A22 Vss Vss A21
100 ... 76
D0 Vss HLDA HOLD Vss NA# READY# Vcc Vcc Vcc Vss Vss Vss Vss CLK2 ADS# BLE# A1 BHE# NC Vcc Vss M/IO# D/C# W/R#
1 ... 25
75 ... 51
A20 A19 A18 A17 Vcc A16 Vcc Vss Vss A15 A14 A13 Vss A12 A11 A10 A9 A8 Vcc A7 A6 A5 A4 A3 A2
TOP VIEW
26 ... 50
LOCK# NC NC NC NC Vcc RESET BUSY# Vss ERROR# PEREQ NMI Vcc INTR Vss Vcc NC NC NC NC NC Vcc Vss Vss Vss
(b)

FIGURE 15-14 (*a*) Pin diagram for 386DX processor view from pin side. (*b*) Top view pin diagram for 386SX processor. (*Intel Corporation*)

is simply an extension of the 8086 and 80286 port structure to include 32-bit ports. Simple 32-bit I/O ports can be constructed by connecting 8-bit I/O port devices such as the 8255A in parallel. A 386 can use an IN or OUT instruction followed by an 8-bit port address to address up to 256 8-bit ports, 128 16-bit ports or 64 32-bit ports. Using the DX register to hold a 16-bit port address, a 386 can access up to 64K 8-bit ports, 32K 16-bit ports, or 8K 32-bit ports.

The PEREQ signal is output by a coprocessor such as an 80387 floating point processor to tell the 386 to fetch the first part of a data word for the coprocessor. The coprocessor will then take over the buses and read the rest of the data word, as we described for the 8087 in Chapter 11. As we also described in Chapter 11, the BUSY# signal is used by the coprocessor to prevent the 386 from going on with its next instruction before the coprocessor is finished with the current instruction. If the ERROR# signal is asserted by a coprocessor, the 386 will perform a type 16 exception.

Regarding the V_{cc} and ground connections, note in Figure 15-15 that the 386 has a large number of V_{cc} pins. It also has a large number of ground connections labeled V_{ss}. These pins are all connected to the appropriate power plane in the PC board.

The RESET, NMI, INTR, HOLD, and HLDA inputs function similarly to the way they do in earlier processors. In a later section we will describe how the 386 handles interrupts while operating in protected mode.

The final group of 386 signals to discuss is the bus control group. The READY# signal is used to insert wait states in bus cycles as needed to interface with slow memory and IO devices.

The BS16# input allows the 386 to work with a 16-bit and/or a 32-bit data bus. If BS16# is asserted, the 386 will transfer data only on the lower half of the 32-bit data bus. If BS16# is asserted and a 32-bit operand is being read from a 16-bit-wide memory, the 386 will automatically generate a second bus cycle to read the second word. For misaligned transfers the 386 will also generate the required number of bus cycles if BS16# is asserted.

The ADS# signal will be asserted when valid addresses, BE signals, and bus cycle definition signals are present on the buses. The 386 address bus is not multiplexed, so an 8086-type ALE signal is not needed. However, in some 386 systems the ADS# signal is used to transfer the address to the outputs of external latches for a

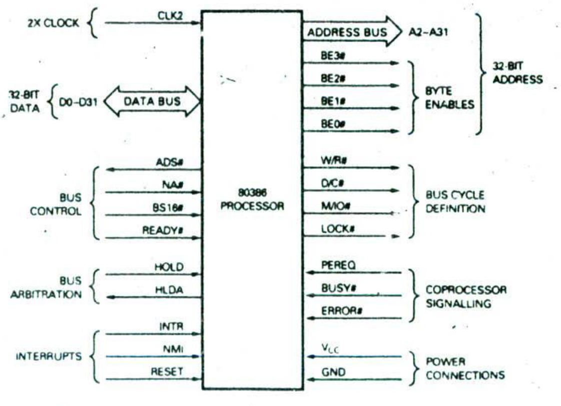


FIGURE 15-15 Signal groups of 386DX. (*Intel Corporation*)

scheme called *address pipelining*. The principle of address pipelining is that if an address is held on the outputs of external latches, the 386 can remove the old address from its address pins and output the address for the next operation earlier in the bus cycle. External control circuitry asserts the next address signal, NA#, to tell the 386 when to output the address for the next operation. Pipelined addressing is not usually necessary in a system with an SRAM cache, because the SRAM cache is fast enough that no wait states are needed.

To help you understand the relationship of some 386 signals, Figure 15-16 shows some 386 nonpipelined read cycles. As you can see, each read operation requires two states, T1 and T2. Note that READY# is made low during T2 so that no wait states are inserted. If the device being read is not fast enough to output data during T2 as required, READY# would be held high longer by external circuitry and a wait state would be inserted in the read cycle after T2.

Incidentally, the 386 contains a large amount of built-in self-test (BIST) circuitry. If the 386 BUSY# input is held low while RESET is held low, the processor will automatically test about 60 percent of its internal circuitry. The self-test requires about 2^{20} CLK2 cycles. If the 386 passes all tests, a "signature" of all 0's will be left in the EAX register.

Now that you have had a short trip around the 386 pins, the next step is to discuss how a 386 can be connected in a system.

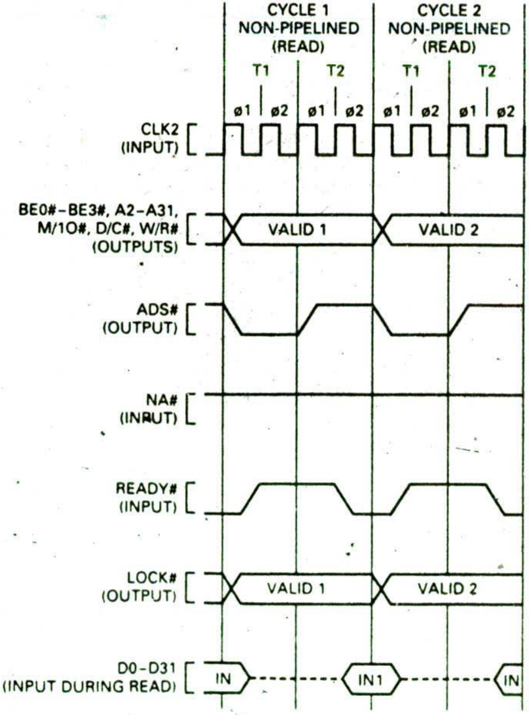


FIGURE 15-16 386 nonpipelined read cycles without wait states. *(Intel Corporation)*

386 System Connections and Interface Buses

THE URDA SDK-386 BOARD

A relatively low cost 386 system useful for prototyping 386-based instruments is the SDK-386 shown in Figure 15-17. This board is similar to the SDK-86 board we discussed in Chapter 7. Both boards are available from University Research and Development, Inc. in Pittsburgh, PA. The SDK-386 board contains a 12-MHz 386, 16 Kbytes of EPROM, 32 Kbytes of static RAM, a keyboard, and a 40-character LCD display. The board also has a serial port and software which allows programs to be developed on a PC-type computer and downloaded to the board for testing and debugging.

This board is useful as a simple, protected-mode learning tool, because the monitor program in ROM on the board runs the 386 in protected mode. The monitor runs as one task and user programs run as another task. A simple keypress allows the user to switch from the user task to the monitor. This feature is very useful for debugging programs and hardware. The documentation for the board shows how the descriptor tables, etc. are set up, and how user programs can call monitor procedures to interface with the keyboard and the display.

386 FULL SYSTEMS

Examples of more complex 386 systems are the IBM PS/2 Model 80, the Compaq SYSTEMPRO 386/33, and

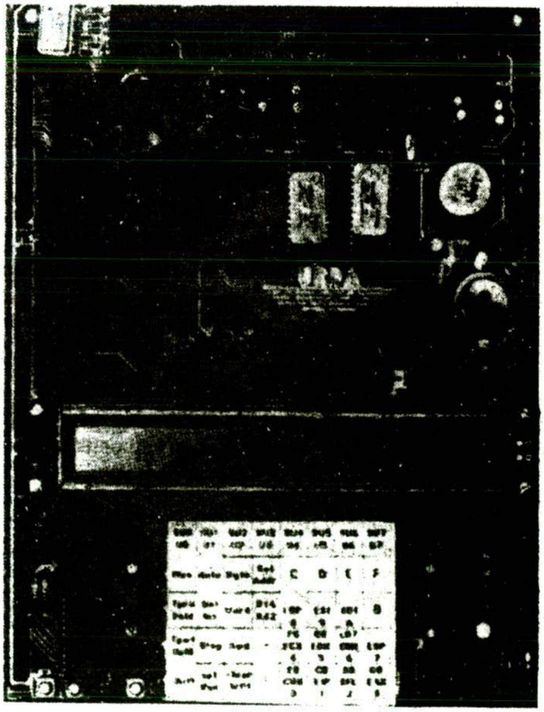


FIGURE 15-17 The SDK-386 prototyping board from University Research and Development Associates.

many similar machines. These systems typically have a megabyte or more of RAM, a 100-Mbyte hard-disk drive, a couple of floppy-disk drives, a VGA CRT controller, parallel ports, and serial ports. In most systems such as this, a 32-bit local data bus is used to interface with the SRAM cache and the DRAM main memory. The 32-bit data bus allows maximum transfer rate between memory and the 386. To interface with the on-board peripheral devices such as timers, priority-interrupt controllers, CRT controllers, serial ports, and parallel ports, the system uses a 16-bit local data bus. A separate I/O expansion bus is used to interface with peripheral boards such as disk controller cards, a network interface card, and a high-resolution graphics card.

Initially we wanted to show you some circuit diagrams for one of these 386 systems, but for several reasons we decided this was not practical. First of all, each of the 40 or 50 different 386 machines currently available uses a different circuit configuration. Second, the motherboards of the newest machines contain mostly large ASICs, which combine many functions in single devices. The Intel 82830, for example, contains an 8-channel DMA controller, a 20-input priority interrupt controller, four programmable timers, wait-state-generating circuitry, a complete DRAM refresh controller, and more. VLSI disk controllers and video controllers are now often included directly on the motherboard of 386 systems. The circuit diagram for a system built with these large ASICs look more like a block diagram than a circuit diagram and shows you little more than you already know about microcomputer structure. Also, most of the manufacturers consider their circuitry proprietary and are unwilling to release diagrams. What we decided would be useful here is to discuss the three bus standards commonly used to interface peripheral boards with the motherboards in these systems.

THE ISA BUS REVIEWED

In Figure 11-7 we showed you the bus used in the original IBM PC and the bus used in the IBM PC/AT. The PC/AT bus is one of the buses used in 386 systems and is now commonly called the *industry standard architecture* or ISA (pronounced e-sah) bus. The other bus schemes commonly used in 386 systems are the *extended industry standard architecture* or EISA (pronounced eye-sah) bus and IBM's *Micro Channel Architecture* or MCA bus.

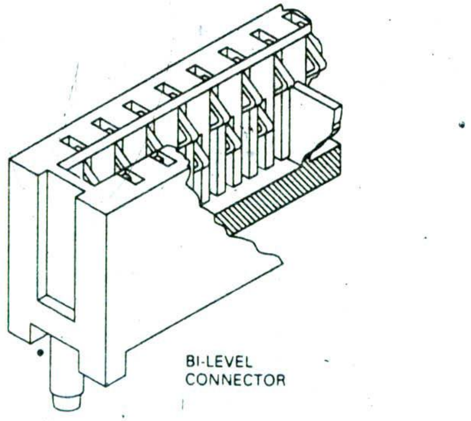
The ISA bus standard has the advantage that many peripheral boards have been developed for it, and competition has kept the price of these boards low. The ISA bus, however, has only 16 data lines and 24 address lines, so it cannot take full advantage of the 32-bit data bus and the 32-bit address bus of the 386. This reduces the speed at which data can be transferred on the bus. Most of the ISA-based 386 machines have partially solved this problem by incorporating up to 16 Mbytes of 32-bit-wide memory and perhaps a cache directly on the motherboard. Since this memory can be accessed directly without going through the peripheral bus, it can operate at the full speed of the 386. In these systems the ISA bus is used only to communicate with peripherals such as disk controller boards, CRT controller boards, network interface boards, etc. Since most of these boards transfer data only 8 bits or 16 bits at a time, the ISA bus limitations do not have an appreciable effect on the performance of a single-user system.

THE EISA BUS STANDARD

For high-performance applications such as network file servers, communication servers, and other multitasking/multiuser systems, the ISA bus does not allow fast-enough data transfer. Also, the ISA bus has no mechanism to arbitrate requests for bus use by "smart" peripheral boards. Systems designed for these applications usually use an EISA or an MCA bus.

As the name indicates, the EISA bus is an extension of the ISA bus to accommodate the needs of the 386 and 486 processors and multimaster systems. It was developed by nine companies as an alternative to IBM's MCA bus standard. The EISA bus uses edge connectors of the same physical size as the ISA bus so that either ISA or EISA boards can be inserted in a slot. However, the EISA connectors have two levels of contacts, as shown in Figure 15-18*a*. If an ISA-based board is inserted, it will go in the connectors only far enough to reach the top level of contacts which contain the ISA bus signals. A notch cut in EISA boards allows them to go into the connectors far enough to also contact the lower level of contacts. These lower contacts contain the additional EISA signals. Figure 15-18*b* shows how these added signal lines are interspersed between the ISA signals. Note the additional address (labeled LA), data, BE#, power, ground, and control pins. The pin positions labeled KEY represent thin slots cut in the PC board so that it aligns properly when inserted. The pins labeled MFG SPEC are used to output a code which identifies the type of board to aid in system configuration during bootup.

In a multiprocessor microcomputer system a processor that can take over the bus to transfer data is called a *bus master*. A board which can take over the bus for a



(*a*)

FIGURE 15-18 (*a*) Two layers of contacts in EISA bus connectors. (*See also next page.*)

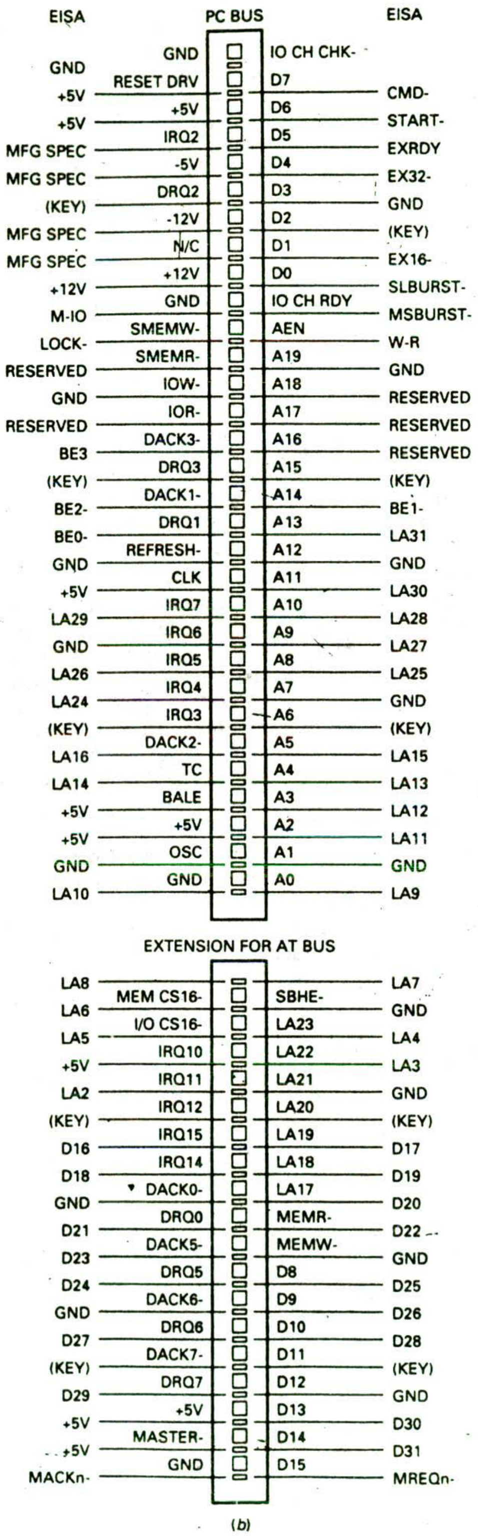


FIGURE 15-18 (*Continued*) (*b*) Pin assignments, EISA bus.

DMA operation is called a *DMA slave*. The EISA bus supports up to six bus masters and 8 DMA slaves. The MACKn and MREQn lines on the EISA bus are used to arbitrate bus requests by multiple masters. These signals are not bused. An individual trace runs from each of these pins to the arbitration logic on the motherboard. The n in these signal names represents the slot number in the system. When a master wants to use the buses, it asserts its MREQ line. If the buses are free and that master is the highest-priority master requesting use of the buses, the arbitration circuitry will assert the MACK signal connected to that master. The master will use the bus for its data transfer. DMA slaves issue requests through the DREQ lines on the bus and receive control from the arbitration circuitry through the DACK lines.

Another feature of the EISA bus is that its interrupts can be individually programmed as edge-triggered for compatibility with ISA boards or level-triggered so that they are less susceptible to noise spikes and they can be shared by several sources. EISA boards use level-triggered interrupts, which can be pulled low by any one of several sources. When the CPU detects an interrupt, it polls each board or device to determine the source of the interrupt.

To help implement an EISA bus in a system, Intel makes the 82358 Bus Controller, the 82357 Integrated System Peripheral, and the 82355 Bus Master Interface Controller. Consult the data sheets for these devices to get more detailed information about the operation of an EISA bus.

THE MICROCHANNEL ARCHITECTURE BUS

IBM's MicroChannel Architecture Bus contains the same types of signals and accomplishes the same functions as EISA, but the two are completely incompatible. MCA boards are smaller and use different edge connectors. Figure 15-19, page 522, shows the MCA bus connector types used in the IBM PS/2 Model 80.

The MCA bus is designed to work with peripheral boards that transfer data in 8-bit, 16-bit, or 32-bit words. For 8-bit peripheral boards, just a 46-pin edge connector is used. For 16-bit peripheral boards, an additional 12-pin connector is used. One of the 16-bit slots also has a 20-pin video extension connector. This slot can be used for an 8514/A high-resolution graphics card. For 32-bit boards an additional 44-pin connector is used in place of the 12-pin connector used on 16-bit slots. Each of the 32-bit slots also has an 8-pin "matched-memory" extension connector.

The Model 80 has five 16-bit MCA slots and three 32-bit slots. One of the 16-bit slots is used for an ESDI hard-disk controller. The 32-bit slots can be used for 32-bit memory boards or other 32-bit peripherals. The signals on the matched-memory extension of the 32-bit slots allow the processor to interrogate memory boards to see if they are designed to transfer data faster than the basic bus rate. If they are, other signals on the connector manage transfers.

The MCA bus uses a distributed scheme to arbitrate bus requests by up to 16 bus masters and DMA request sources. To request the bus, one or more masters assert

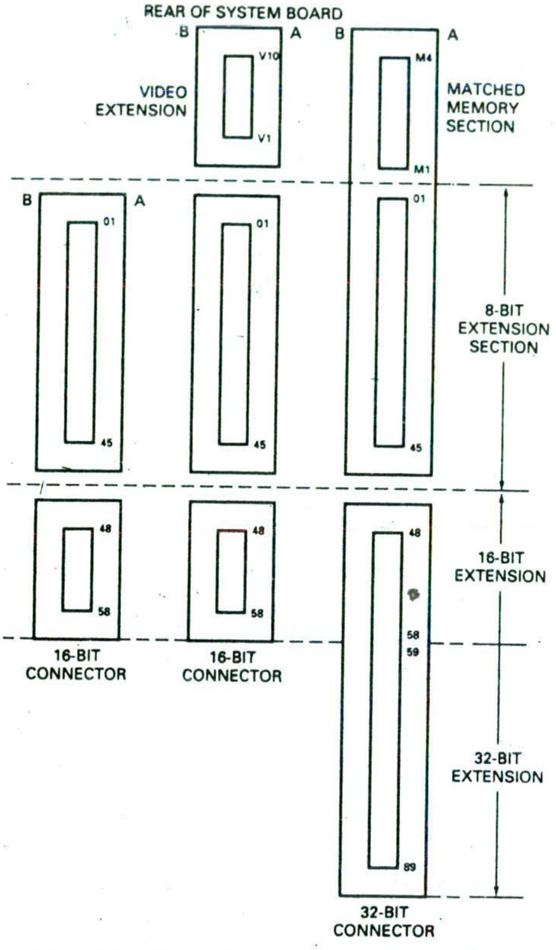


FIGURE 15-19 IBM's MicroChannel Architecture bus connector types.

the PREEMPT line to the central control circuitry low. At the appropriate time the control circuitry drives the ARB/GNT line high. The arbiter on each master then asserts its arbitration code on the ARB0–ARB3 lines. If an arbiter sees a code that is lower than its code, it removes its arbitration signals. This means that the master with the lower arbitration code assumes control of the bus. To signal the arbitration is complete, the central control point asserts the ARB/GNT signal low. Incidentally, the interrupt lines on the MCA bus are level-triggered.

Now that you have had a brief introduction to the system connections and buses used in 386 systems, let's take a look at the internal architecture of the device and talk about the different 386 operating modes.

Real Operating Mode

A 386 can operate in real mode, protected mode, or a variation of protected mode called *virtual 8086* mode. After a reset the 386 operates in real address mode. In this mode it functions basically as a fast 8086 or real-mode 80286. The register set for the 386 in real mode in a superset of the 8086 and 80286 real-mode register sets. As shown in Figure 15-20, the 32-bit general-purpose registers are referred to as extended AX or EAX, EBX, ECX, EDX, etc. Instructions can, for example, refer to AL, AH, AX, or EAX. The assembler automatically codes the instruction for the register size referred to in an instruction.

The 386 in real mode computes memory addresses using the same segment base and offset mechanism used by the 8086. For this mode only the selectors or visible parts of the segment registers are used. Note that the 386 has two additional data segment registers, FS and GS, so programs can have up to four data segments. The length of segments in 386 real mode is fixed at 64 Kbytes, and any attempt to access a location outside a segment will cause a type 13 exception.

The address range of 386 real mode is limited to 1 Mbyte, so address lines A20–A31 are normally all low. The only exception to this is that during a reset these address lines are all made high to access the boot ROM at the highest locations in the 32-bit address space of the 386. As soon as the boot-ROM code does a far jump or call, the A20–A31 lines will go low and stay low as long as the 386 is in real mode. A 386 in real mode uses the address space 00000–003FFH for the interrupt-vector table and services interrupts in the same way as an 8086 does.

One new feature of the 386 is the debug registers shown in Figure 15-20. A software debugger can load breakpoint addresses in these registers to aid in debugging. A 386 can be instructed to "break" when the address unit in the processor computes a linear address which matches one of the addresses in the debug registers. The older method of setting a breakpoint involved replacing an instruction with a breakpoint instruction such as INT 3. This method, of course, cannot be used to debug code in ROM, but the breakpoint register method can because it does not depend on modifying code bytes.

The 32-bit EFLAGS register in the 386 is an extension of the 16-bit registers in the 8086 and 80286. For future reference the upper right corner of Figure 15-20 shows the names of the bits in the EFLAGS register, but in real mode only the lower 16 bits have meaning.

The final real-mode registers to note in Figure 15-20 are the control registers CR0–CR3. The lower 16 bits of CR0 correspond to the machine status word (MSW) of the 80286. As with an 80286, a 386 is switched to protected-mode operation by setting the LSB of this register to a 1. Register CR1 is reserved by Intel, and registers CR2 and CR3 are used for paged mode functions, which we discuss later.

386 Protected-Mode Operation

INTRODUCTION

The real power of a 386 lies in its protected-mode and virtual 8086-mode features. These features are designed in a very versatile way, so that almost any conceivable

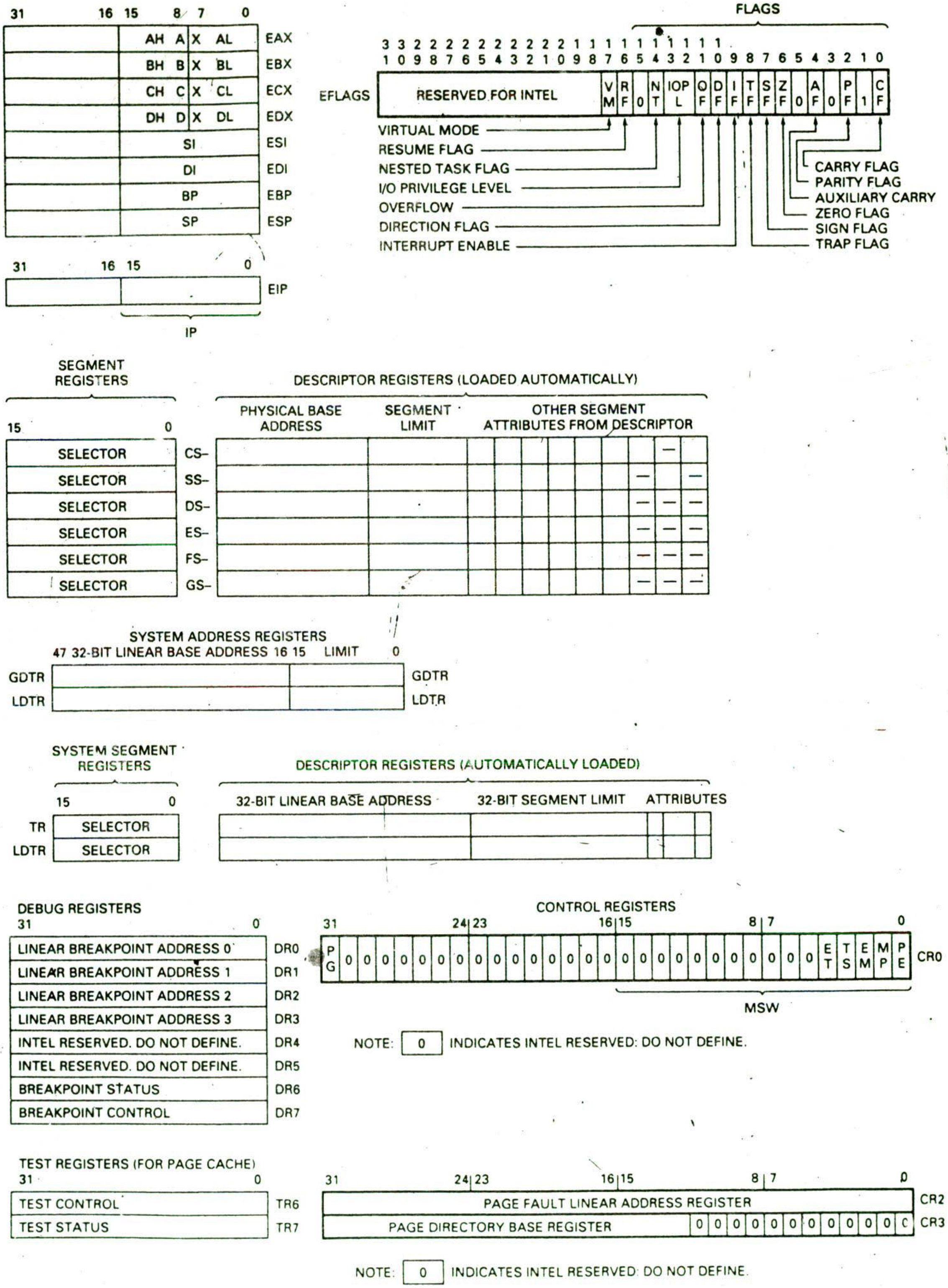


FIGURE 15-20 Intel 386 microprocessor register set. *(Intel Corporation)*

operating system or program can be implemented on a 386. The problem with this versatility is that it leads to an almost unbelievable amount of detail in a complete description of how the 386 operates in these modes. In reality, unless you are writing a 386-based operating system, you can probably live a very happy life without knowing all these details. In the following sections we have tried to give just enough details so that you can understand the basic protected-mode operation of a 386, how its features fit the needs of a multiuser/multitasking operating system, and how programs are written for a 386. If you need to know all the minute details, consult the Intel 80386 Programmer's Reference Manual and the Intel 80386 System Software Writer's Guide.

As you read through the following sections, the key concepts you should try to fix in your mind are: how a 386 computes physical addresses in segments-only mode and in paged mode, how a 386 provides protection for operating system code and protection for user tasks, the basic operation of a gate, the protected-mode interrupt response, the task switch process, and the operation of the "flat" system model.

SEGMENTATION AND VIRTUAL MEMORY

As we said in the preceding section, a 386 is switched from real mode to protected mode by setting the LSB of the CR0 register. The virtual memory addressing scheme of a 386 in protected mode is very similar to that of the 80286 we described earlier, except that 386 segments can be much larger and an optional paging mechanism allows segments to be divided into 4-Kbyte pages for faster swapping in and out of physical memory.

In protected mode each 386 address consists of a 16-bit segment selector and a 32-bit offset. As we described earlier in a section on virtual memory, the selector points to a descriptor for the segment in a table of descriptors and the offset specifies the location of the desired code or data in the segment. Using a 32-bit offset value means that segments can be anywhere from 1 byte in length to 2^{32} or about 4 Gbytes in length.

Figure 15-21 shows the format for 386 segment selectors and how these selectors are used to access a descriptor in a descriptor table. The 13-bit index part of this selector is multiplied by 8 and used as a pointer to the desired descriptor in a descriptor table. The index value is multiplied by 8 because each descriptor requires 8 bytes in the descriptor table. Among other things the descriptor contains the physical base address for the segment. The MMU adds the base address from the descriptor to the effective address or offset part of the logical address from the instruction to produce the physical memory address.

There are two major categories of descriptor table in

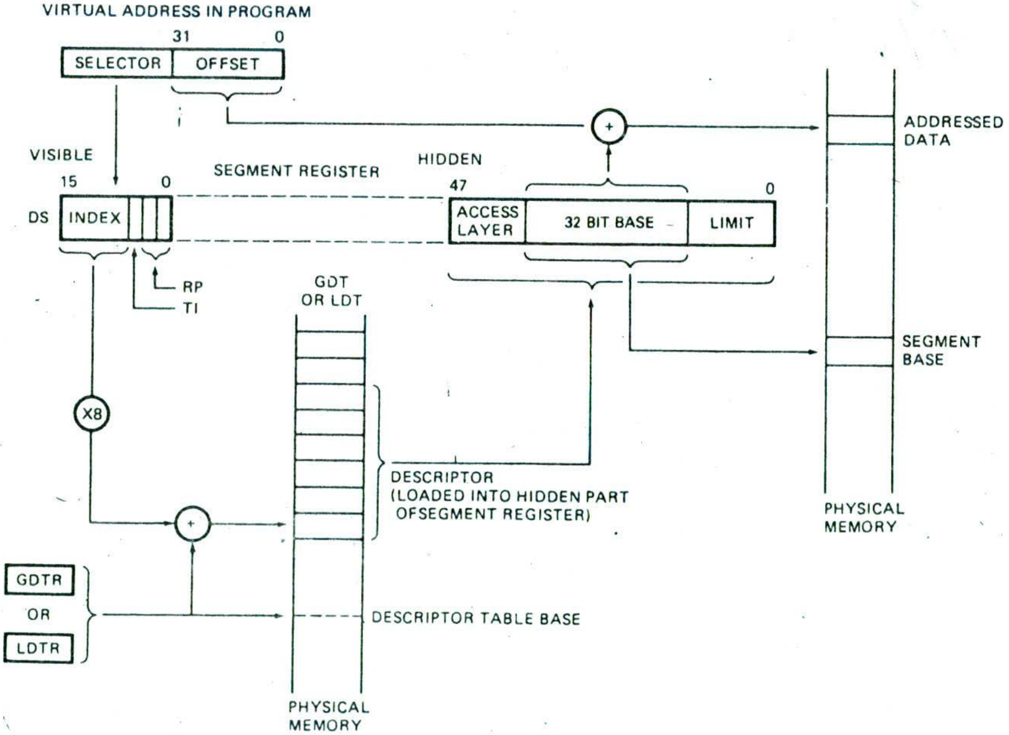


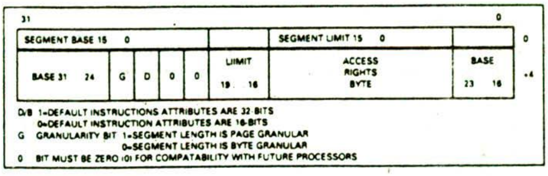
FIGURE 15-21 Diagram showing how the 386 uses a selector to access a descriptor in a descriptor table and how it computes the physical (linear) address. *(Intel Corporation)*

a 386 system, global and local. A system has only one *global descriptor table* or GDT. The GDT contains, among other things, the segment descriptors for the operating system segments and the descriptors for segments which need to be accessed by all user tasks. A *local descriptor* table or LDT is set up in the system for each task or closely related group of tasks. Figure 15-22 shows, in diagram form, how this works. Tasks share a global descriptor table and the memory area defined by the descriptors in it. Each task can have its own local descriptor table and memory area defined by the descriptors in it. Setting up individual LDTs protects tasks from each other because one task cannot access the LDT of another task.

If the *table indicator* bit (bit 2) of a segment selector is a 0, then the upper 13 bits will index a segment descriptor in the global descriptor table. If the TI bit of the selector is a 1, then the upper 13 bits of the selector will index a segment descriptor in a local descriptor table.

The least significant 2 bits of a segment selector, the *requested privilege level* or RPL bits, are part of the 386's built-in protection features, which we discuss later. For now, let's take a closer look at segment descriptors.

Figure 15-23*a* and Figure 15-23*b* show the formats for the 386 segment descriptors and the access rights byte of the descriptors. First notice in the descriptor that 32 bits are set aside for the segment's physical base address and 20 bits are set aside for the size or limit of the segment. If you remember that we said 386 segments can be up to 2^{32} bytes long, you may wonder why only 20 bits are set aside here for the size of the segment. The answer to this is that if the granularity or G bit in the descriptor is a 0, the 20-bit limit value represents the length of the segment in bytes. With a 0 value in the G bit, then, a segment can be up to 1 Mbyte in length. If the G bit in the descriptor is a 1, the 20-bit limit value represents the length of the segment in 4-Kbyte blocks. The maximum limit value of 1,048,576 blocks × 4 Kbytes/block then gives a maximum segment length of 4 Gbytes. If an attempt is made to access a location outside the specified limit for a segment, a type 5 exception will be produced. This mechanism prevents a program from accessing memory outside its defined segments.



31
0
SEGMENT BASE 15 0
SEGMENT LIMIT 15 0
0
BASE 31 24
G
D
0
0
LIMIT 19 16
ACCESS RIGHTS BYTE
BASE 23 16
+4
D/B 1=DEFAULT INSTRUCTIONS ATTRIBUTES ARE 32-BITS
0=DEFAULT INSTRUCTION ATTRIBUTES ARE 16-BITS
G GRANULARITY BIT 1=SEGMENT LENGTH IS PAGE GRANULAR
0=SEGMENT LENGTH IS BYTE GRANULAR
0 BIT MUST BE ZERO (0) FOR COMPATABILITY WITH FUTURE PROCESSORS

(*a*)

	BIT POSITION	NAME	FUNCTION	

	7	PRESENT (P)	P=1 SEGMENT IS MAPPED INTO PHYSICAL MEMORY
			P=0 NO MAPPING TO PHYSICAL MEMORY EXITS. BASE AND LIMIT ARE NOT USED	

	6 - 5	DESCRIPTOR PRIVILEGE LEVEL (DPL)	SEGMENT PRIVILEGE ATTRIBUTE USED IN PRIVILEGE TESTS	

	4	SEGMENT DESCRIPTOR (S)	S=1 CODE OR DATA (INCLUDES STACKS) SEGMENT DESCRIPTOR
			S=0 SYSTEM SEGMENT DESCRIPTOR OR GATE DESCRIPTOR	

TYPE FIELD DEFINITION	3	EXECUTABLE (E)	E=0 DATA SEGMENT DESCRIPTOR TYPE IS	IF DATA SEGMENT (S = 1, E = 0)

	2	EXPANSION DIRECTION (ED)	ED=0 EXPAND UP SEGMENT, OFFSETS MUST BE ≤ LIMIT
			ED=1 EXPAND DOWN SEGMENT, OFFSETS MUST BE > LIMIT	

	1	WRITEABLE (W)	W=0 DATA SEGMENT MAY NOT BE WRITTEN INTO
			W=1 DATA SEGMENT MAY BE WRITTEN INTO	

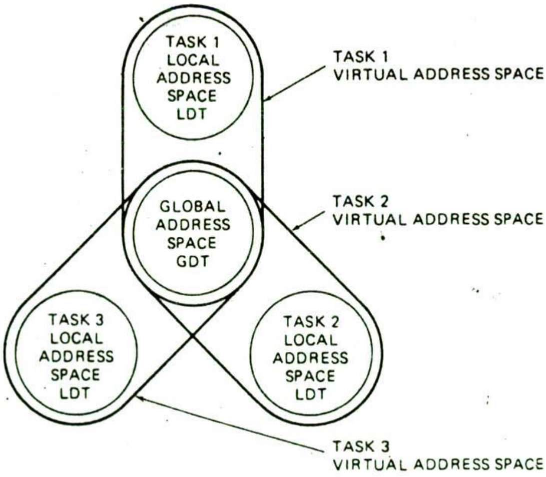
	3	EXECUTABLE (E)	E=1 CODE SEGMENT DESCRIPTOR TYPE IS	IF CODE SEGMENT (S = 1, E = 1)
	2	CONFORMING (C)	C=1 CODE SEGMENT MAY ONLY BE EXECUTED WHEN CPL ≥ DPL AND CPL REMAINS UNCHANGED	

	1	READABLE (R)	R=0 CODE SEGMENT MAY NOT BE READ
			R=1 CODE SEGMENT MAY BE READ	

	0	ACCESSED (A)	A=0 SEGMENT HAS NOT BEEN ACCESSED
			A=1 SEGMENT SELECTOR HAS BEEN LOADED INTO SEGMENT REGISTER OR USED BY SELECTOR TEST INSTRUCTIONS	

(*b*)

FIGURE 15-23 (a) 386 descriptor format. (*b*) Access rights byte format for code and data segment descriptors. (*Intel Corporation*)



TASK 1 LOCAL ADDRESS SPACE LDT
TASK 1 VIRTUAL ADDRESS SPACE
GLOBAL ADDRESS SPACE GDT
TASK 2 VIRTUAL ADDRESS SPACE
TASK 3 LOCAL ADDRESS SPACE LDT
TASK 2 LOCAL ADDRESS SPACE LDT
TASK 3 VIRTUAL ADDRESS SPACE

FIGURE 15-22 Diagram showing how tasks can be isolated from each other by having separate local descriptor tables but can share a common global descriptor table. (*Intel Corporation*)

Byte 5 of a descriptor, the access byte, contains information about the privilege level, access, and type of the segment. To give you an idea of the kind of information contained in the access byte of a descriptor, Figure 15-23*b* summarizes the meanings of the bits in the access bytes of code segment and data segment descriptors. Skim through the descriptions to get an overview. Note the P bit, which is used to indicate whether the segment is present in physical memory, the privilege-level bits, which specify the privilege level that a program must have to access the segment, and the A bit, which is set if the segment has been accessed. The operating system can periodically read and reset the A bit to determine how often the segment has been accessed. A segment which has not been recently used can be swapped out to disk when space for a new segment is needed.

When a program attempts to access a segment, the selector for the segment is loaded into the visible part of the segment register. To access a data segment, for example, the selector might be loaded into the visible part of the DS, ES, FS, or GS register. When the selector is loaded into the visible part of the segment register, the descriptor for the segment is automatically loaded into the hidden part of the segment register or segment descriptor cache, as it is commonly called. If the privilege level of the selector and the privilege level of the current code segment is not as high (or is higher than) the privilege level of the descriptor, an exception will be produced and the access will not be allowed. If the privilege level is high enough, the P bit in the descriptor will be checked to see if the segment is present in physical memory. If the segment is not present, a segment-not-present (type 11) exception will be generated and the exception handler will read the segment in from disk to physical memory. Once the segment is in physical memory, the address unit computes the physical addresses as needed to access the data words in the segment. As shown in Figure 15-21, the offset from the original address is added to the segment base address from a descriptor to form a linear address. For a 386 operating in segments-only mode, this linear address is the physical address that will be output on the address and BE lines to memory.

To complete the general picture of how a 386 manages virtual segments, we simply need to show you how it keeps track of where the descriptor tables are in memory. The 386 keeps the base addresses and limits for GDT and LDT descriptor tables currently being used in internal registers. The *global descriptor table register* (GDTR) shown in the middle of Figure 15-20 is used to hold the 32-bit base address and limit for the global descriptor table. This register is initialized with a load global descriptor table register (LGDT) instruction when the system is booted. The *local descriptor table register* (LDTR) shown in Figure 15-20 is used to hold the base address and limit of the local descriptor table for the task currently being executed. The LLDT instruction is used to load this register. The LLDT instruction can be executed only by programs executing at the highest privilege level. Therefore, unless a task is operating at the highest privilege level, it cannot intentionally or maliciously access the local descriptor table of another task. Task switching is usually handled by the operating system kernel, which operates at the highest-priority level.

386 SEGMENT PRIVILEGE LEVELS AND PROTECTION

When an attempt is made to access a segment by loading a segment selector into the visible part of a segment register, the 386 automatically makes several checks. First of all, it checks to see if the descriptor table indexed by the selector contains a valid descriptor for that selector. If the selector attempts to access a location outside the limit of the descriptor table or the location indexed by the selector in the descriptor table does not contain a valid descriptor, then an exception is produced.

The 386 also checks to see if the segment descriptor is of the right type to be loaded into the specified segment register cache. The descriptor for a read-only data segment, for example, cannot be loaded into the SS register, because a stack must be able to be written to. A selector for a code segment which has been designated "execute only" cannot be loaded into the DS register to allow reading the contents of the segment.

If all these protection conditions are met, the limit, base, and access rights byte of the segment descriptor are copied into the hidden part of the segment register. The 386 then checks the P bit of the access byte to see if the segment for that descriptor is present in physical memory. If it is not present, a type 11 exception is produced. The exception-handler procedure for this exception will swap the segment into physical memory, set the P bit in the descriptor, and restart the interrupted instruction.

After a segment selector and descriptor are loaded into a segment register, further checks are made each time a location in the actual segment is accessed. An attempt to write to a code segment or a read-only data segment, for example, will cause an exception. Also, the limit value contained in the segment descriptor is used to check that an address produced by program instructions does not fall outside the limit defined for the segment.

User tasks can be protected from each other in a 386 system by giving each task its own local descriptor table. The LDT register, which points to a user's local descriptor table, can only be changed with the LDTR instruction or by a task switch. The LDTR instruction can be executed only at the highest privilege level, which is usually reserved for the operating system. Likewise, a switch from one user task to another is done by the operating system at the highest privilege level, so user tasks operating at lower privilege levels cannot cause switches to other user tasks. Also, because of limit checking, a task cannot accidentally or intentionally access descriptors in another task's local descriptor table.

System software, such as the operating system kernel, is protected from corruption in several ways. One way we have already mentioned is that code segments can be made "execute only" so that they cannot be written to. The second and most important way that the operating system can be protected is with privilege levels. Figure 15-24 illustrates how a 386 protected-mode system can be set up with four privilege levels. As we mentioned before, the operating system kernel is assigned the highest privilege level, which is privilege level 0. System services such as BIOS procedures might be run at privilege level 1, and custom device drivers, etc. might operate at privilege level 2. Application programs and user tasks are usually operated at the privilege level 3, the lowest level.

The privilege level for a segment is represented by bits 5 and 6 of the access byte in the segment descriptor. (See Figure 15-23*b* for access byte format.) These 2 bits are referred to as the *descriptor privilege level* or DPL. This privilege level is established when the program is built.

The privilege level of an executing task is represented by the DPL bits in the access byte of the descriptor currently in the CS descriptor cache. This privilege level is referred to as the *current privilege level* or CPL.

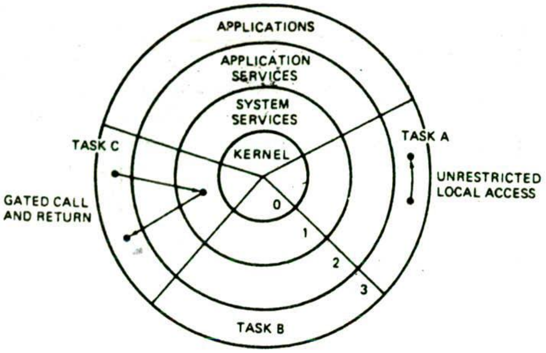


FIGURE 15-24 Diagram showing how a 386 system can be set up with four privilege levels. (*Intel Corporation*)

When a program needs to access a data segment, it does so by loading a segment selector into, for example, the visible part of the DS register. The privilege level encoded in the least significant bits of this selector is referred to as the *requesting privilege level* or RPL.

To successfully access a segment, both the RPL and the CPL must be a number less than or equal to the DPL of the segment. In other words, the privilege level of the currently executing task and the privilege level of the requesting selector must both be greater than or the same as the privilege level of the desired segment in order for access to be granted. If these conditions are not met, then an exception will be generated. The point here is that normally a task cannot directly access a segment which has a higher DPL.

CALL GATES

The question that might come to mind at this point is, If a task cannot access a segment with a more privileged DPL, how can user programs access the operating system kernel, BIOS, or utility procedures in segments which have more privileged DPLs? The answer to this is that a procedure located in a segment which has a higher privilege level can be called indirectly through a special structure called a *gate*. There are four types of gates: call, trap, interrupt, and task. For now, we will just describe how a call gate operates.

A gate is simply a special type of descriptor. Gate descriptors are put in the GDT or in an LDT, just as segment and other descriptors are. When a program does a call to a procedure in another segment, the selector for that segment's call gate is loaded into the CS register, and the call gate descriptor is loaded into the hidden part of the CS register. The call gate descriptor contains a selector which points to the descriptor for the segment where the procedure is actually located. The call gate descriptor also contains the offset of the called procedure in its segment.

If the call is determined to be valid, then the selector from the call gate and the corresponding segment descriptor will be loaded into the CS register. The processor then uses the base address from the segment descriptor and the offset value from the call gate descriptor to compute the physical address of the called procedure. Therefore, the call is done indirectly, through the call gate descriptor, rather than directly through a segment descriptor.

This indirect access has two major advantages. First, this approach permits another level of privilege checking before access to the procedure in the higher-privileged segment is allowed. The privilege level of the calling program is compared with the privilege level specified in the call gate. If the privilege level of the calling program is lower than the privilege level specified in the call gate, the access will not be allowed. If, for example, the DPL in the call gate descriptor is 2, a level 2 program can use the call gate to call a privilege level 1 procedure, but a level 3 program cannot.

Another advantage of the indirect call gate approach is that user programs cannot accidentally enter higher-privileged segments at just any old point. If they are going to enter at all, they must enter at the specific offsets contained in the call gate descriptors. This is similar to the type of protection provided by using software interrupts to call BIOS and DOS functions instead of calling them directly.

I/O PRIVILEGE LEVELS

When a 386 is operating in protected mode, the 386 has two mechanisms for protecting I/O ports. The first mechanism involves the I/O privilege-level bits in the 386 EFLAGS register shown in Figure 15-20. Only the operating system or a procedure operating at a privilege level 0 can set these IOPL bits. In order to execute the IN, INS, OUT, OUTS, CLI, and STI instructions, the CPL of a procedure or task must be the same or a lower number than IOPL represented by these bits. If a procedure does not meet the IOPL test, a privilege-level exception will be generated.

The second mechanism for protecting ports from unauthorized access is an optional I/O permission bit map which allows ports to be associated only with specific tasks. If this feature is used, a map is set up for each task. Each bit in the map represents a byte-wide port address, so 16-bit ports use 2 bits each and 32-bit ports use 4 bits each. A 0 in a map bit means the port is available to the task.

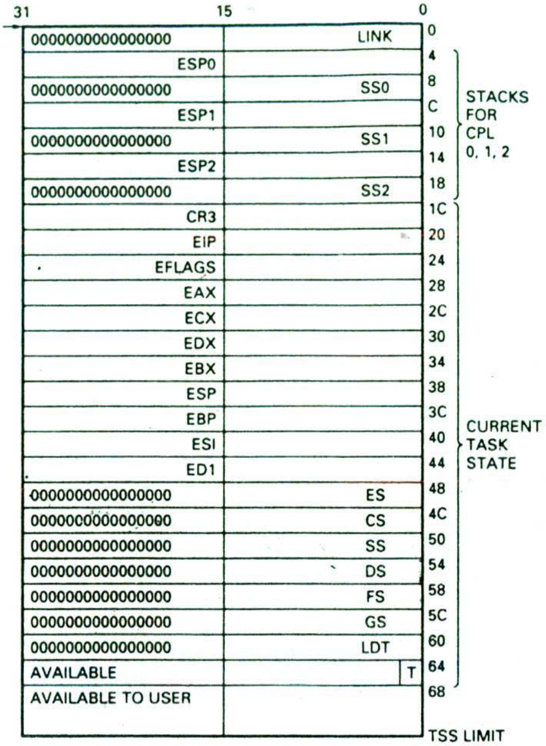
When a task attempts to access a port, the 386 first compares the CPL of the task with the IOPL. If the access passes the IOPL test and an I/O bit map is in force, the 386 will then check the map bits corresponding to the addressed port. If the map has a 0 in the bit(s) for that port, access will be granted. If not, an exception will be generated. Incidentally, when a 386 is operating in real address mode, none of the port protection mechanisms are in effect.

INTERRUPT AND EXCEPTION HANDLING

For operation in protected mode, gate descriptors for the interrupt and exception procedures are kept in a special descriptor table called the interrupt descriptor table or IDT. This table can be located anywhere in memory. During initialization the base address and limit for the interrupt descriptor table are loaded into the *interrupt descriptor table register* (IDT) shown in Figure 15-20 with an LIDT instruction.

When an interrupt or exception occurs, its type is multiplied by 8 and added to the IDT base address in the IDT register. The result is a pointer to a gate descriptor in the interrupt descriptor table. The gate here can be an interrupt gate, a trap gate, or a task gate.

An interrupt gate, for example, contains a selector for the segment where the interrupt procedure is located, not the base address of the segment. The reason for this is so that the privilege level can be checked before access to the interrupt procedure is granted. If the CPL is high enough, then the selector from the gate will be loaded into the CS register and used to access the descriptor for the segment containing the interrupt procedure. The segment descriptor can be in the LDT or GDT. The 32-bit offset from the gate will be added to the base address from the descriptor to produce the linear address for the actual interrupt procedure. This is basically the same mechanism we described previously for the operation of a call gate, except that at the end of the procedure an IRET instruction is used instead of an RET. Incidentally, an interrupt procedure that needs to be accessible from any privilege level is put in a code segment that is made "conforming" by setting bit 2 in the access byte of its descriptor.



31	15	0	
0000000000000000	LINK	0	
ESP0		4	STACKS FOR CPL 0, 1, 2
0000000000000000	SS0	8	
ESP1		C	
0000000000000000	SS1	10	
ESP2		14	
0000000000000000	SS2	18	
CR3		1C	CURRENT TASK STATE
EIP		20	
EFLAGS		24	
EAX		28	
ECX		2C	
EDX		30	
EBX		34	
ESP		38	
EBP		3C	
ESI		40	
ED1		44	
0000000000000000	ES	48	
0000000000000000	CS	4C	
0000000000000000	SS	50	
0000000000000000	DS	54	
0000000000000000	FS	58	
0000000000000000	GS	5C	
0000000000000000	LDT	60	
AVAILABLE	T	64	
AVAILABLE TO USER		68	
		TSS LIMIT	

FIGURE 15-25 386 task state segn.ent format. (*Courtesy Intel Corporation*)

TASK SWITCHING

In a multiuser operating system each user's program can be set up as a separate task. When a user's time slice is up, the operating system switches execution from the current user's task to the next user's task. A similar process takes place in a single-user system which is operating in a multitasking mode. As we pointed out earlier, one of the main concerns in a multitasking system is saving the state or context of a task so that it will continue execution properly when it gets another time slice.

Each task in a 386 protected-mode system is assigned a *task state segment* or TSS. Figure 15-25 shows the format for a 386 TSS. As you can see, the TSS holds copies of all registers and flags, the selector for the task's LDT, and a link to the task state segment of the previously executing task. Descriptors for the task state segments are kept in the global descriptor table, where they can be accessed by the operating system during a task switch. The *task register* (TR) in the 386 holds the selector and the descriptor for the task state segment of the currently executing task. The *load task register* (LTR) instruction can be used to load the task register with the selector and segment descriptor for a specific task, but during a task switch the task register is automatically loaded with the selector and descriptor for the new task.

A task switch may be done in any one of four ways:

1. A long jump or call instruction contains a selector which points at a task state segment descriptor. The call instruction is used if a return to the previously executing task is desired. A jump instruction is used if a return to the previously executing task is not desired. This is the simplest method and can be easily implemented by the operating system kernel at the end of a time slice.
2. The selector in a long jump or call instruction points to a task gate. In this case the selector for the destination TSS is in the task gate. The indirect mechanism here is similar to that we described above for call gates and has the same advantages regarding privilege levels and protection.
3. An interrupt occurs, and the interrupt selector points to a task gate in the interrupt descriptor table. The task gate contains the selector for the new task state segment. If the access passes all the privilege level tests, the selector and descriptor for the interrupt task will be loaded into the task register. The nested task (NT) bit in the EFLAGS register will be set.
4. An IRET instruction is executed with the NT bit in the EFLAGS register set. Complex interrupt procedures are often written and managed as separate tasks. The IRET instruction uses the back link selector in the task state segment to return execution to the interrupted task. This is similar to the way the IRET instruction works in real-mode operation.

We don't have space or inclination to explain the details of all the possible task switch scenarios, but we will make a few comments about the CALL/JMP method.

When a far CALL or JMP is executed to switch tasks, the privilege levels are first checked. As with any far call or far jump instruction, the RPL of the CALL selector and the CPL of the executing program must both be less than or equal to the DPL of the desired segment, or an exception will be produced.

Assuming proper privilege levels, the 386 will check if the task state segment for the new task is present in physical memory and generate a not-present exception if it is not. If necessary, the exception handler will load the TSS for the new task.

The 386 then copies all the register values for the current task to its task state segment. The value copied for the EIP is the offset of the next instruction after the one that caused the task switch.

At this point the old TSS is no longer needed, so the 386 loads the task register with the selector and the descriptor for the TSS of the new task. The 386 then automatically copies all the values for the new task from its TSS to the 386 registers. Execution then continues using the segment and offset values copied from the TSS.

In a multiuser/multitasking system, the operating system might use a JMP instruction to switch from the operating system task to a user task. A clock tick will interrupt the processor at the end of the time slice. If the interrupt descriptor table contains a task gate which points to the operating system task, then the state of the current task will be saved in its TSS, and execution will switch to the operating system task. The operating system can use another JMP instruction to switch to the next user's task.

PAGING MODE

The protected-mode segmentation and virtual memory scheme we described for the 386 in the preceding section is essentially the same as that for the 80286. The main difference is that 386 segments can be as large as 4 Gbytes, instead of only 64 Kbytes. The designers of the 386 realized that the time required to swap very large segments in and out of physical memory would be too long, so they added an optional paging mechanism to the design of the 386. The paging mechanism allows segments to be divided into 4-Kbyte pages for faster swapping.

The 386 is switched into paging mode by setting the MSB of the CR0 register with a simple MOV CR0, EAX-type instruction. In this mode the paging unit in the 386 uses the linear address, computed by the segmentation unit as described above, to produce the physical address. Figure 15-26 shows how this paging scheme works. To help fix it in your mind, we will first explain the paging scheme from the bottom up and then from the top down.

The Intel data sheets refer to each 4-Kbyte page in physical memory as a *page frame*. The least significant 12 bits of the linear address from the segmentation unit represent the offset of the desired data word within a 4-Kbyte page frame. The 32-bit base addresses and some other information for up to 1024 page frames are kept in a *page table* in memory. For future reference Figure 15-27*a* shows the details of the 4-byte entry placed in a page table for each page frame. The 10 address bits, A12–A21, from the linear address are used to select the desired entry in one of the page tables.

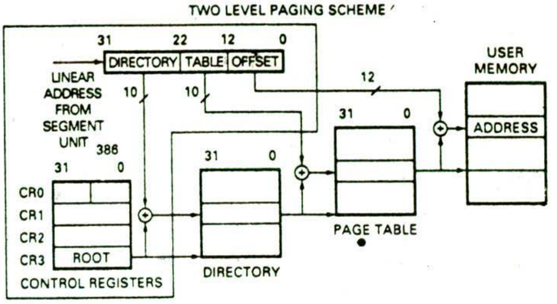


FIGURE 15-26 Diagram showing how a 386 computes physical addresses when paging mode is enabled.

A system can contain up to 1024 page tables. The 32-bit base addresses and some other information for the page tables are kept in another table called the *page directory*. The format for the 4-byte entries in the page directory is the same as the format shown for a page table entry in Figure 15-27*a*. The 10 address bits, A22–A31, are used to select the desired entry in the page directory. The 32-bit base address for the page directory is kept in control register 3 (CR3) in the 386.

Looking at this from the top down then, CR3 points to the base of the page directory and linear address bits A22–A31 point to one of 1024 possible entries in the page directory. The selected entry in the page directory points to the base address of one of up to 1024 page tables, and linear address bits A12–A21 point to one of the entries in the selected page table. The selected entry in the page table contains the 32-bit base address of the desired 4-Kbyte page frame. Linear address bits A0–A11 are used to access the desired code or data word in the selected page frame. These bits are added to the base address from the page table entry to produce the physical

31 . 12	11 10 9	8	7	6	5	4	3	2	1	0
PAGE TABLE ADDRESS 31 . . . 12	OS RESERVED	0	0	D	A	0	0	U — S	R — W	P

(*a*)

U/S	R/W	PERMITTED LEVEL 3	PERMITTED ACCESS LEVELS 0, 1, OR 2
0	0	NONE	READ/WRITE
0	1	NONE	READ/WRITE
1	0	READ-ONLY	READ/WRITE
1	1	READ/WRITE	READ/WRITE

(*b*)

FIGURE 15-27 (*a*) Format for 386-page directory and page table entries. (*b*) Access rights produced by combinations of R/W and U/S bits in 386-page table entries.

address that will be output to memory. The maximum amount of memory represented by this structure is 1024 page tables × 1024 pages/page table × 4096 bytes/page, or about 4 Gbytes, the full 32-bit address space of the 386. A system can be set up with just one page directory, but a more common practice is to give each task its own page directory and, thereby, its own set of page tables. Later we show you how the 386 task switch mechanism makes provisions for easily switching page directories.

As we said before, the page directory is located in memory and the page tables are located in memory. To avoid having to read page directory entries and page table entries from memory tables during each memory access, the 386 maintains a special cache called a *translation lookaside buffer* or TLB. The TLB is a four-way set-associative cache which holds the page table entries for the 32 most recently used pages. (Refer to the discussion of caches in Chapter 11 if the term set-associative is a little rusty in your mind.) When the 386 generates a linear address, the upper 20 bits of that address are compared with the tags for the 32 entries in the TLB. If there is a match, the page table entry for the desired page is in the TLB. The base address from this entry is used to compute the physical address. If there is no match, the 386 reads the page table entry from memory and puts it in the TLB. If the P bit in the page table entry is a 1, indicating that the page is present in physical memory, then the physical address will be computed and the desired word in the page accessed. If the P bit in the page table entry is a 0, indicating that the page is not present in physical memory, the processor will generate a page fault exception (type 14). After the page fault exception handler swaps the page into physical memory, the paging unit will compute and output the physical address for the desired word.

When the 386 paging mode is enabled, the U/S and R/W bits in the page directory entries and the page table entries can be used in place of or in addition to the segmentation protection mechanisms. The U/S bit in a directory or page table entry is used to specify one of two privilege levels, user or supervisor. A 0 in the U/S bit specifies the user privilege level, which corresponds to segment privilege level 3, the lowest level. A 1 in the U/S bit specifies supervisor privilege level, which corresponds to segment privilege levels 0, 1, and 2.

The R/W bit in a page directory or page table entry can be used to establish read-write access rights for pages or page tables. Figure 15-27*b* shows the access rights produced by various combinations of U/S and R/W. Note that for these bits 11 represents the most privilege and 00 the least privilege. If the access rights specified in a page directory entry are different from the access rights specified in a page table entry, the least privileged of these determines the access rights.

SUMMARY OF MEMORY MODELS

The memory in a 386 or 486 system can be set up using the segments-only model, the segmented-paged model, the simple flat model, or the paged flat model.

We thoroughly described the segments-only model in a previous section. This is the only protected-mode memory model available on an 80286. Versions 1.1 and 1.2 of Microsoft's OS/2 protected-mode operating system were designed to run on an 80286 system, so they use this model.

As we explained before, 386 segments can be too large to be conveniently swapped in and out of memory, so the 386 allows a paging mechanism to be switched in after the segmentation unit. The paging unit divides segments into 4-Kbyte pages for swapping in and out of physical memory. This segmented-paged model allows a programmer to think in terms of logical segments and the virtual memory hardware to think in terms of easily moved pages. However, one problem with this combined approach is that the amount of time required to manage all the descriptor tables, segments, page tables, and pages in a complex system becomes too large. A second problem is that developing the software to manage all this is a complex task. Also, the amount of memory used by all the tables can become excessively large. For these and other reasons, Microsoft's OS/2 for the 386, Novell's Netware 386, and many other programs for 386 and 486 systems use the flat memory model, which effectively removes segmentation.

The 386 does not have a way to turn off segmentation, but you can effectively eliminate segmentation by initializing all the segment registers with the same base address and initializing the segment limits for 4 Gbytes. Each segment then corresponds to the 4-Gbyte physical address space of the 386. The 32-bit offset or effective address part of each memory address is large enough to access any location in this 4-Gbyte space. The different parts of programs are simply located at different offsets in the address space.

This memory mode is referred to as the *simple flat system model* and is useful for dedicated control applications that need the fastest possible task switching and don't need all the segment-based protection features. The SDK-386 board we discussed earlier uses the simple flat model, and as we show later, this makes program development for it quite easy. Also, the flat system model makes it easy to transport software written for nonsegmented devices such as those in the Motorola 68000 family devices to a 386.

Paged flat model systems enable the 386 paging mechanism to provide virtual memory-management and protection features. The present (P) bit in a page directory entry indicates whether the requested page table is present in memory and the P bit in the page table entry indicates whether the requested page is present in memory. The accessed (A) bit in a page table entry indicates whether the page has been accessed. The operating system can periodically check and reset this bit to determine how often the page is being used. If the page has not been used lately, it can be replaced when the operating system needs space for a new page. The dirty (D) bit in a page table entry will be set if data has been written to the page. In this case the operating system must write the modified page out to disk before swapping a new page into its space. As we discussed earlier, the user/supervisor (U/S) bit in the page directory entries and the page table entries provide two privilege levels. The read/write (R/W) bit in a page table entry allows a page to be marked as read only or read/write. The I/O permission bit map which we mentioned earlier can provide protection for I/O ports.

The point of all this is that the paged flat memory model provides fast virtual memory capability and a degree of protection adequate for most applications.

386 Virtual 8086-Mode Operation

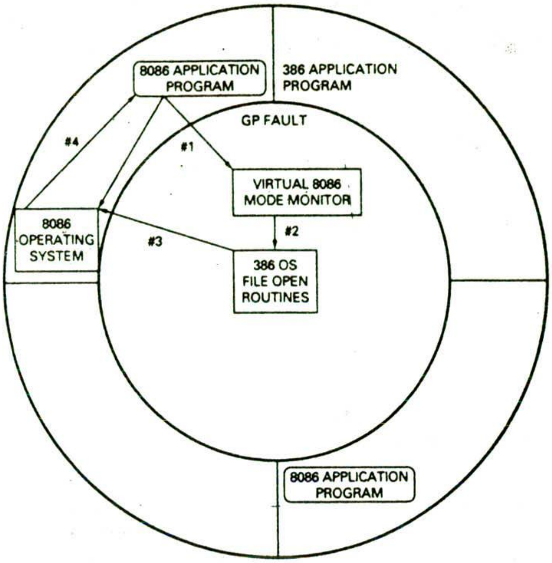
As we pointed out in an earlier discussion, it is difficult to switch a 286 processor back and forth between real and protected mode. This limitation makes a 286 hard to use for a multitasking system, which must run a mixture of tasks that use segment-offset addressing and protected-mode tasks that use descriptors. The 386 virtual 8086 mode solves this problem. A 386 operating in protected mode can easily switch to virtual 8086 mode to execute a time slice of an 8086-type program and then easily switch back to protected mode to execute a time slice of a protected-mode task. This means that some users in a multiuser system can be running programs under protected mode UNIX V and other users can be running real-mode DOS programs.

When a 386 operating in protected mode does a task switch, it examines the VM bit in the EFLAGS register. If this bit is set, the 386 will enter virtual 8086 mode to execute the new task. If the VM bit is not set, the 386 will execute the new task as a normal protected mode task.

In virtual 8086 mode the 386 computes physical addresses using the segment-offset mechanism used by an 8086. Therefore, the address range of a virtual 8086 mode task is 1 Mbyte. For a single virtual 8086 task this address range is in the lowest 1 Mbyte in the processor address space. If a system needs to run several different 8086 type tasks, then the 386 is operated in paging mode so that each 8086 task can be given a different page table and a different set of pages in physical memory. A side benefit of using the paging mode is that the U/S and R/W bits in the page directory entries and the page table entries provide protection that is normally not available in real mode.

In order to run virtual 8086 mode tasks, the operating system must have a section of privilege level 0 code called a *virtual machine monitor*. The main purpose of this monitor is to intercept interrupts, exceptions, and INT n instructions which occur during the execution of the 8086 task. Figure 15-28 shows how this works for an INT n instruction.

As you well know from previous chapters, most 8086 system programs use INT n software interrupts to access BIOS and DOS I/O procedures. In virtual 8086 mode the INT n instruction can be executed only at privilege level 0, the highest privilege level. Since an 8086 task always operates at level 3, the lowest privilege level, the 386 will generate an exception whenever the 8086 program executes an INT n instruction. The handler for this exception is in the virtual machine monitor, so the monitor effectively takes over execution at this point.



8086 APPLICATION MAKES "OPEN FILE CALL" → CAUSES GENERAL PROTECTION FAULT (ARROW #1)
VIRTUAL 8086 MONITOR INTERCEPTS CALL. CALLS 386 OS (ARROW #2)
386 OS OPENS FILE RETURNS CONTROL TO 8086 OS (ARROW #3)
8086 OS RETURNS CONTROL TO APPLICATION. (ARROW #4)
TRANSPARENT TO APPLICATION

FIGURE 15-28 Operation of virtual machine monitor when 8086 virtual mode application program makes DOS call to open a file. *(Courtesy Intel Corporation)*

During the task switch to the monitor, the state of the 8086 task is saved in its TSS. Also the VM bit in the EFLAGS register is reset, so the monitor can operate in normal protected mode.

If the call was to a function such as the DOS "open file" command, the monitor will call the equivalent procedure in the 386 protected-mode operating system to open the file. This mechanism maintains all the protection built into the main 386 operating system. When the file has been opened, execution is returned to the DOS operating system. The IRET instruction used to return to the virtual 8086 DOS program restores the 8086 task state. As part of this, the VM bit in the EFLAGS register is restored to a 1 so that the 8086 program restarts in the virtual 8086 mode. For other DOS function calls which do not involve I/O, the monitor may return execution to DOS to perform the function. After the function is completed, DOS returns execution to the 8086 program.

In virtual 8086 mode interrupts are also intercepted by the monitor. In most cases the monitor will transfer execution to the 386 protected mode operating system to service the interrupt. To service interrupts the 386 operating system uses the interrupt descriptor table and gate scheme we described previously, so protection is maintained. If protection is not an issue, the monitor may return execution to DOS or to the 8086 program to service the interrupt.

When a clock tick interrupt occurs to signal the end of a time slice, execution will switch from the 8086 task to the monitor task. The monitor task through an IDT gate will switch to the 386 operating system scheduler. The scheduler will then switch to the next user task. The VM bit in the EFLAGS register image of the TSS for the new task will determine whether the task is executed in virtual 8086 mode or 386 protected mode. The point here is that the 386 provides a relatively simple mechanism to alternate between 8086-type programs and 386 protected-mode programs.

Now, before we dig into 386 instruction set enhancements and programming, let's summarize what we have found out about the 386 so far.

Summary of 386 Hardware and Operating Modes

The 386 is a 32-bit processor which is upward compatible from the 8086, 80186, and 80286. In real address mode the 386 functions as a fast 8086 and uses the segment-offset address mechanism to address 1 Mbyte of memory.

In its protected mode a 386 can address 4 Gbytes of physical memory and 64 Terabytes of virtual memory. Each protected-mode address consists of a 16-bit selector and a 32-bit offset or effective address. The 32-bit offset component means that segments can be as large as 4 Gbytes. An optional paging mechanism allows segments to be broken into 4 Kbytes pages for faster swapping in and out of memory. The 386 uses the 16-bit selector to access the descriptor for the segment in the global descriptor table or in a local descriptor table. The segment base address from the descriptor is added to the 32-bit offset to produce the linear address. In segments-only mode, the linear address is the physical address. If paging is enabled, the paging unit uses the linear address, a page directory, and a page table to produce the physical address.

The 386 contains several mechanisms to protect OS code from user tasks and user tasks from each other. One of these mechanisms is privilege levels. The operating system code is given a privilege level of 0, the highest privilege level, and user code is given a lower privilege level. Any direct attempt by a program to access a code or a data segment with a higher privilege level will generate an exception. Programs can, however, access procedures at a higher privilege level through an indirect method called a gate. The gate allows a second check on the privilege level of the access and makes sure the access is to the correct location in the procedure. A second protection mechanism is bounds checking. Any attempt to access a location outside the limit specified for a segment in its descriptor will generate an exception.

For a 386 operating in protected mode, interrupts are vectored through gates in the interrupt descriptor table. This indirect approach allows interrupt procedures to be protected.

In a 386 system using the flat memory model, the entire physical memory is treated as a single large segment. All segments are given the same base address and limit, so they share this segment. The 32-bit offset contained in every memory address is large enough to access any location in the 4-Gbyte physical address space of the 386. In a larger system using the flat memory model, paging is enabled so that virtual memory and protection can be implemented.

When the 386 does a protected-mode task switch, it automatically copies the state of the current task to a task state segment created for that task and loads the state of the new task from its TSS. If the 386 finds the VM bit of the EFLAGS register set when it does a task switch, the 386 goes to virtual 8086 mode. In this mode the 386 can directly execute 8086 type programs which use segment-offset addressing. The interrupt at the end of a time slice will cause the 386 to switch back to full protected mode so the operating system can switch to the next task using protected-mode features.

386 Instruction Set Additions and Enhancements

A SECOND LOOK AT THE 386 REGISTER SET

In Figure 15-20 we showed you that the 386 register set is a superset of the 8086 and 80286 register sets. The 386 register-type instructions allow you to specify 8-bit registers and 16-bit registers as you do in 8086 instructions or to specify 32-bit registers. In 386 instructions you can specify, for example, AH, AL, AX or EAX as an operand. The instruction MOV EAX,EBX, for example, will copy the 32-bit number in the extended BX register to the extended AX register. You cannot copy an 8-bit part of a register to a 32-bit register with an instruction such as MOV EBX,AL. Also, you cannot directly access just the upper 16 bits of a 32-bit register. If you need to copy just the upper 16 bits of, for example, the EAX register into the BX register, you can first rotate the upper 16 bits of EAX into the lower 16 bits with the ROR EAX,16 instruction and then use MOV BX,AX. If you need to put EAX back in its initial condition, you just do another ROR EAX,16 instruction. The 386 contains a "barrelshifter," which can shift an operand any number of bits in one clock pulse, so these rotates do not appreciably slow the overall operation. Incidentally, even a 386 real-mode program which uses 16-bit segments can use the 32-bit extended registers for data operations.

For real-mode programs only the lower 16 bits of the extended instruction pointer (EIP) are used, because only 16 bits are needed to access any location in a 64-Kbyte real-mode segment. In a protected-mode program a code segment can be specified as 16-bit or 32-bit. To specify a code segment as 16-bit, you simply write the term USE16 after SEGMENT in the segment declaration line. The line CODE SEGMENT USE16, for example, declares a 16-bit segment named CODE. A segment is specified as 32-bit by putting the term USE32 after SEGMENT in the segment declaration. If the segment is specified as a USE16 segment, then the maximum segment limit is 64 Kbytes, and only the lower 16 bits of EIP will be used to access instruction bytes. If the segment is specified as USE32, then the maximum segment limit is 4 Gbytes, and all 32 bits of EIP are used to hold the offset of an instruction byte.

The 386 contains two new segment registers, FS and GS, which can be used as additional data segments. None of the 386 instructions use these segments as their default segment, so you usually have to use a segment override prefix on an instruction which accesses a data item in one of these segments. We will show you how to do this in a later program example.

NEW ADDRESSING MODES AND SCALING

When an 8086 executes the instruction MOV AX,PATIENT_RECORD [BX] [DI], it computes the effective address of the memory operand by adding a displacement represented by the name PATIENT_RECORD, an offset contained in the BX register, and an index value contained in the DI register. For an 8086 only BX and BP can be used as base registers in this way, and only SI and DI can be used as index registers. A 386 can use any of the eight 32-bit, general-purpose registers as a base register, and it can use any of the 32-bit, general-purpose registers except ESP as an index register. When a 386 executes the instruction MOV BX, [EAX+EDX], for example, it will compute the effective address of the memory operand by adding the 32-bit number in EAX to the 32-bit number in EDX. Note that these new addressing modes work only with the 32-bit extended registers. You can't, for example, use just AX as a base pointer.

The 386 also has another powerful addressing feature called *index scaling*, which is useful for accessing successive elements in an array of words, double words, or quad words. Index scaling allows the value contained in an index register to be automatically multiplied by a specified scale factor of 2, 4, or 8 when an instruction executes. If a 386 executes the instruction MOV EAX, [EBX+EDI*4], for example, it will multiply the index value in EDI by a scale factor of 4 and add the result to the value from the EBX register to produce the effective address. The double word pointed to in DS by this effective address will be copied into the EAX register. If this instruction is part of a loop which processes an array of double words, then all that you have to do to get ready for the next trip around the loop is to increment the index value in EDI. When the MOV instruction is executed again, the new index value will automatically be multiplied by the scale factor in computing the effective address.

Even though real-mode data segments can be only 64 Kbytes long, you can still use index scaling and these new 32-bit addressing as long as the effective address produced does not exceed 16 bits. In a later section we show you an example program which demonstrates how to do this.

NEW INSTRUCTIONS

The 386 instruction set includes all the 8086/80186/80286 instructions and extends these instructions to work with 32-bit data words and 32-bit offsets. The 386 also includes several new instructions. In this section we briefly explain the functions of these new instructions and give you an example of each. Then we make a few comments about 8086 instructions that have been enhanced in the 386.

Bit Scan and Test Instructions

BSF—Bit scan to the left until nonzero bit is found.

EXAMPLE:

```
BSF CX, DX     ; Scan DX to left until nonzero bit,
               ; leave bit number in CX
               ; Zero flag set if all DX=0
```

BSR—Bit scan to the right until nonzero bit is found.

EXAMPLE:

```
BSR CX, DX     ; Scan DX to right until nonzero bit,
               ; leave bit number in CX
               ; Zero flag set if all DX=0
```

BT—Bit test and put specified bit in carry flag.

EXAMPLE:

```
BT EBX, 4      ; Copy bit 4 of EBX to carry flag
```

BTC—Bit test and complement.

EXAMPLE:

```
BT EBX, 7      ; Copy complement of bit 7 to CF
```

BTR—Bit test and reset.

EXAMPLE:

```
BTR WORD PTR [BX], 3 ; Bit 3 of [BX] to CF
                     ; Reset bit 3 of [BX]
```

BTS—Bit test and set.

EXAMPLE:

```
MOV CL, 4
BTS EAX, CL    ; Copy bit 4 of EAX to CF
               ; Set bit 4 of EAX
```

Data-type conversions

CDQ—Convert signed double word in EAX to quadword in EDX:EAX.

CWDE—Converts signed word in AX to double word in extended EAX.

Segment load instructions

These instructions are similar to LDS and LES instructions described in Chapter 6.

LFS—Load FS segment register and specified base register with values from specified memory locations.

EXAMPLE:

```
LFS BX, DWORD PTR [DI]
                        ; Load FS and BX with
                        ; DWORD from memory at [BX]
```

EXAMPLE:

```
LFS EBX, FWORD PTR [DI]
                        ; Load FS with 16 bit
                        ; selector and EBX with 32-bit
                        ; offset for memory at [DI]
```

LGS—Load GS segment register and specified register from specified memory locations.

LSS—Load SS segment register and specified register from specified memory locations.

Move and expand instructions

MOVSX—Move and sign extend to fill destination register.

EXAMPLE:

```
MOVSX CX, BL      ; Copy BL to CL, extend sign bit of
                  ; BL through all of CH
```

MOVZX—Move and zero extend to fill destination register.

EXAMPLE:

```
MOVZX CX, BL      ; Copy BL to CL, fill CH with zeros
```

Set memory flag word instruction

SETxx—Set all bits in specified byte if condition xx is met. xx here can be any condition from conditional jump mnemonics.

EXAMPLE:

```
SETC TooBig       ; Set all bits in flag TooBig if
                  ; Carry flag set
```

Shifts between words

SHLD—Shift specified number of bits left from one operand into another.

EXAMPLE:

```
SHLD EAX, EBX, 8  ; Shift upper 8 bits from EBX
                  ; into lower 8 bits of EAX
                  ; EBX unchanged
```

SHRD—Shift specified number of bits right from one operand into another.

EXAMPLE:

```
SHRD EAX, EBX, 8  ; Shift lower 8 bits from EBX
                  ; into upper 8 bits of EAX.
                  ; EBX unchanged
```

INSTRUCTION ENHANCEMENTS

Several of the 386 instructions have significant improvements over the 8086/80186/80286 versions. Here are a few notes about these improvements.

1. The 386 string instructions work with double-word operands as well as with word and byte operands. A "B" at the end of an instruction mnemonic specifies byte operands, a W specifies word operands, and a D specifies double-word operands. Examples are CMPSB, CMPSW, and CMPSD.
2. The destination for a 386 conditional jump can be anywhere in the segment containing the jump instruction. Conditional far jumps must still be done by changing the jump condition and using an unconditional far jump as we showed you for the 8086 in Chapter 4.
3. The LOOP instructions can use the CX register or the ECX register as a counter. If you want CX to be used, write the instructions as LOOPW, LOOPWE, and LOOPWNE. If you want the ECX register to be used as the counter, write the instructions as LOOPD, LOOPDE, and LOOPDNE.
4. PUSHFD pushes the 32-bit EFLAGS register and POPFD restores it.
5. PUSHAD pushes the 8 general-purpose 32-bit registers on the stack, and POPAD restores these registers except for the value of ESP which is ignored.
6. IRETD pops the double word EIP, a double word for CS, and the EFLAGS register off the stack. The high word of the value popped for CS is discarded.
7. The IMUL instruction can now perform signed multiplication on any general-purpose register and a memory location or another general-purpose register.
8. In addition to the protected-mode instructions inherited from the 80286, the 386 instructions used to move data to/from the control registers (CR0–CR3), the debug registers (DR0–DR7), and the test registers (TR0–TR7) can be executed only in protected mode at privilege level 0. These instructions are simple MOV register, register instructions.

386 Programming

INTRODUCTION

The tools and techniques used to write a program for a 386 or a 486 depend very much on whether it is a system program or an application program and whether it utilizes protected mode or not. The tools and techniques are also determined by whether the program is going to execute in a graphical user-interface environment such as Microsoft's Windows 3.0 or OS/2. In this section we give you an introduction to writing 386 programs for five different programming environments.

386 PROGRAMS FOR MS DOS–BASED SYSTEMS

Current versions of DOS are designed to run on 8086/8088, real-mode 80286, or real-mode 386/486 systems. If you want a program to be able to run under DOS on any one of these systems, then you have to write it for the "weakest link" in the group, the 8086. The 8086 and C programming examples throughout this book were in fact compiled and run on a 386-based system. If you are writing a program that you are sure will only be run under DOS on a 386- or 486-based machine, you can write the program to take advantage of the 32-bit processing capability, addressing modes, and enhanced instructions of the 386. Assuming that you are using the MASM or TASM assembler, you tell the assembler to accept 386 instructions by putting the .386 directive at the top of your source program, as shown in Figure 15-29*a*.

If you are using the simplified segment directives, make sure to put the .386 directive after the .MODEL directive, as shown in Figure 15-29*b*. This order tells the assembler to create 16-bit segments which are compatible with real-mode operation. If you put the .386 directive before the .MODEL directive, the assembler will create 32-bit code and data segments which can be used only in protected mode.

Even though 386 real-mode programs are limited to 64-Kbyte segments and unless bank switching is implemented, to a 640-Kbyte address space, you can still use the 32-bit extended registers, addressing modes, and new instructions of the 386. The simple example program in Figure 15-29*a* shows some of the possibilities. The main points we included in this example are: how to declare segments; how to access data in the new segments, FS and GS; and how to use the 32-bit addressing modes and scaling.

First note that the code and data segments are specified as 16-bit with USE16 directives. The USE16 on the data segment directive specifies a maximum segment length of 64 Kbytes as required for real-mode operation. The USE16 on the CODE SEGMENT line tells the assembler to compute all memory addresses using the segment base and 16-bit offset method.

Next in the example note that we assume and initialize the FS register, just as we did the DS, ES, and SS registers in earlier program examples. The first action of the program then shows you how to read a double word from this segment to the 32-bit EAX register. A segment override prefix must always be used for references to the FS or GS segment, because there is no default as there is with DS. The ROR EAX,16 instruction rotates the 32-bit EAX register around 16 bits to the right. As we said earlier, if execution is limited to a 386 system, the 32-bit registers can be used for data operations, even in real mode.

```
 Some 386 addressing modes and instructions

 .386
DATA    SEGMENT USE16
        BIGVAL  DD 12345678H
        TABLE   DW 4235H, 7590H, 4968H, 3817H
DATA    ENDS

CODE    SEGMENT USE16
        ASSUME CS:CODE; FS:DATA

START: MOV  AX, DATA                ; Initialize FS
       MOV  FS, AX                  ;  register
;Read 32-bit operand from memory
       MOV  EAX,FS:BIGVAL           ; Get double word
       ROR  EAX, 16                 ; Swap word order
       MOV  FS: BIGVAL, EAX         ; Put back result
;Process table
       MOV  CX, 04
       MOV  EDI, 0
NEXT:  MOV  DX,FS:[TABLE+EDI*2]     ; Word from array to DX
                                    ; Scan from left for
       BSF  AX, DX                  ; first zero bit
       JNZ  MORE                    ; Skip if all zeros
       MOV  FS:[TABLE+EDI*2],AX     ; Store number of
                                    ;  first zero bit
MORE:  INC  EDI                     ; Increment index
       LOOP NEXT
       MOV  AX, 4C00H               ; Return to DOS
       INT  21H
CODE   ENDS
       END START
```

(a)

```
;Simplified segment directives example
; for 16-bit segments

DOSSEG
 .MODEL large
 .386
 .DATA

 TABLE DW 4235H, 7590H, 4968H, 3817H

 .CODE
 START:  MOV AX, DATA          ; Initialize FS register
         MOV FS, AX
```

(b)

FIGURE 15-29 (*a*) 386 real-mode program using traditional segment directives. (*b*) Simplified segment directives for generating 16-bit 386 segments.

The next section of the example in Figure 15-29*a* shows how the 32-bit addressing modes can be used to help process an array of words. The instruction MOV DX,FS: [TABLE + EDI*2] copies a word from the array to DX. The effective address for this instruction is computed by multiplying the contents of EDI by 2 and adding the result to the displacement represented by the name TABLE. The first time through the loop EDI contains 0, so the effective address is just TABLE, the offset of the first word in the array. Before the loop executes again EDI is incremented to 1. During the next execution of the MOV DX,FS: [TABLE+EDI*2] instruction the effective address will be TABLE+2, the offset of the second word in the array. An important point here is that in real-mode operation an exception will be generated if the effective address produced by an instruction is greater than 64 Kbytes.

Within the loop we use one of the new 386-bit instructions to process the word read in from memory. The BSF AX, DX instruction will scan the DX register starting from the left until it finds a nonzero bit. The number of the first nonzero bit will be loaded into AX. If all bits in DX are zeros, the zero flag will be cleared. In this case we just leave the word of all zeros in the array and process the next word. If the word is not all zeros, we write the number of the first nonzero bit in the memory word and then process the next word.

Finally, in the example program we use a familiar DOS function call to return execution to DOS. This is not a particularly significant program, but it does show you a little of what you can do with the added features of the 386 if you are willing to sacrifice 8086 downward capability.

386 PROGRAMS FOR THE SDK-386

As we told you earlier, you can download the binary programs from a PC- or PS/2-type computer to an URDA SDK-386 board through an RS-232C link. The SDK-386, board operates in protected mode using the simple flat memory model, so you can use it to experiment with a simple, dedicated 386 system such as those in Chapter 10.

During initialization the monitor program sets up a global descriptor table, a local descriptor table, and an interrupt descriptor table. The GDT, LDT, and IDT registers in the 386 are loaded with the base addresses and limits for these tables. The monitor also sets up a system task state segment and a user task state segment. As part of the initialization the selector for the user task state segment is loaded into the backlink field of the system TSS. When a user presses the RUN or the STEP key, an IRET instruction causes a task switch to the user task. This executes the user program. If the user presses the BREAK key, an NMI interrupt will be generated. This causes a task switch to the system task so that registers, memory locations, etc. can be examined. The BRPT key can be used to load up to four breakpoint addresses in the 386 debug registers. As we explained earlier, the 386 checks each memory address and will break if it finds any of the specified breakpoint addresses.

You have considerable flexibility in how you write a program for the SDK-386 board. The simplest approach is to use a format slightly modified from that in Figure 15-29*a*. The flat model memory mode used by this board means that all segments start from absolute address 00000000H, and all the segment registers contain the same base address. System and user programs use 32-bit offsets to access code and data words in this shared segment. The contents of the segment registers then do not have to be changed during a switch from the system task to the user task. This has important implications for how you write a program to run on the board.

The D bit in the code segment descriptor determines whether the 386 uses 16-bit effective addresses or 32-bit effective addresses. If the D bit is a 0, then 16-bit effective addresses are produced and if D = 1, 32-bit addresses are produced. The monitor program in the board sets the D bit of the code segment descriptor to a 1, so this means that 32-bit addressing is assumed. To make your program compatible, you make the code segment a USE32 type so that the assembler will produce 32-bit offsets. Your data segments should also be made USE32 type to be compatible with the segments set up by the monitor. Incidentally, you don't have to initialize data segments as part of your program, because the monitor loads the selectors and descriptors for these, and they are not changed when execution switches to the user task.

The user area of RAM on the board begins at 300H, so you should include an ORG 300H directive before the code segment in your program. The program will then be assembled to run in this address space in RAM. After the program is assembled and linked, it can be downloaded to the board and run.

For a more complex program the board allows you to use the segment-based protection features of the 386. The global descriptor table contains four user-definable descriptors and the local descriptor table contains six user-definable descriptors. The interrupt descriptor table also contains a user-definable descriptor for the USER INTERRUPT key on the board and another user-definable descriptor. We don't have a space here to show you how, but you can use these descriptors to define custom segments for your programs.

WRITING A 386 PROTECTED-MODE OPERATING SYSTEM

In the unlikely case that you should have to write a 386 protected-mode operating system or monitor program, you should be aware of Intel's 386 Relocation, Linkage, and Library (RLL) tools which run on IBM PC/AT or newer microcomputers. This tool set contains a *binder*, which is a high-powered linker that can combine object modules compiled from different languages into tasks, combine segments, resolve PUBLIC/EXTERNAL references, assign virtual addresses, and generate a file which can be loaded into RAM for debugging. The tool set also contains utilities for working with library functions. Another important part of the tool set is the *builder*, which allows a programmer to assign physical addresses to segments; set segment access rights and limits; create gates; create global, local, and interrupt descriptor tables; create task state segments; and set up the boot process.

Incidentally, the Intel 80386 System Software Writer's Guide shows a simple example of a flat system and a simple example of a segmented system.

MICROSOFT'S OS/2 2.0 OPERATING SYSTEM

As we told you earlier, MS DOS is for the most part a single-user, single-task operating system and does not take advantage of the virtual memory and multitasking capabilities of the 286/386/486 processors. The probable successor to DOS is Microsoft's OS/2, which is a single-user, multitasking operating system. Microsoft's OS/2 version 1.0 was an early attempt at a multitasking operating system for the 80286 processor. OS/2 1.0 and the later versions of OS/2 for the 80286 can multitask several protected mode tasks and one real mode task. The real-mode task is run in a "DOS compatibility box." The reason that only one real mode task can run is the difficulty in switching an 80286 from protected mode to real mode and back. The user interface for OS/2 1.0 was a typed command line similar to DOS.

The next version of OS/2, OS/2 1.1 introduced a new graphical user interface (GUI) called the *Presentation Manager* or PM. PM is similar to the screen-based interface you may have seen on Apple Macintosh computers. In PM you execute commands by using a mouse to move a cursor to a desired command in a menu of commands or to an *icon* which represents the command. You then execute the command by clicking a key on the mouse. PM also allows you to have multiple "windows" open on the screen. You can "cut" something from one window and "paste" it into another window. The file manager in PM allows you to display a directory tree on the screen, select a file from the tree, and perform some action on the file by just moving cursor around on the screen and clicking the mouse key at the appropriate points.

OS/2 1.2 kept Presentation Manager and added the High Performance File System (HPFS). Instead of the FAT used by the DOS file system, the HPFS uses a different system which allows much faster file access. Another obvious improvement in HPFS is that filenames can be longer than 8 characters. HPFS also sets up an "extended attribute" block for each file. The operating system or an application program can use this block to describe and control use of the file.

OS/2 version 2.0, designed to run only on 386 and 486 systems, uses the virtual 8086 mode of these processors to implement Multiple Virtual DOS Machines (MVDM) capability. In addition to the features of earlier versions, OS/2 2.0 can multitask any mixture of DOS programs, applications written for earlier versions of OS/2, and applications written specifically for version 2.0. OS/2 2.0 uses the flat paged memory model that we described earlier for the 386.

In the preceding chapters we showed you how to use DOS function calls to open files, read files, etc. in your programs. In OS/2 2.0 there are three different *application program interfaces* (APIs) or sets of functions which can be called to perform these functions. One is the real-mode DOS compatible API which is used by the programs operating in a DOS compatibility box. The functions in this API are called with software interrupts such as INT 21H. The second API contains the 16-bit functions that are compatible with OS/2 1.x application programs. To use one of these functions the required parameters are first pushed on the stack, and then the procedure is called by name. The third API contains the 32-bit functions used for OS/2 2.0 applications. The functions in this API are also called by name after pushing the required parameters on the stack. Unlike the 16-bit API calls, the parameters for these calls are pushed on the stack in the same order as parameters for C function calls are pushed. In fact, the 32-bit API is in some ways like an extension of the C Run Time Library you met in Chapter 12. One major difference between the 32-bit API and the C RTL is in the way the functions are connected to the .exe program.

When a C program is linked, the object code files for library functions are linked with the object code for the compiled C program modules to make the .exe program. The OS/2 APIs are dynamic link libraries (DLLs). When a program containing an API call is linked, a reference to the function is put in the .exe file instead of the object code for the called function. When the program is loaded into memory to be run, the library containing the function is loaded into memory where the program can access it. This approach may seem strange at first, but it has several advantages. First, the .exe programs are much smaller, because they do not contain the large library functions. This means that the .exe files take less space on a hard disk. Also, the API functions are reentrant, so a library can be shared by multiple tasks on a multitasking system. This saves memory, because each task does not have to have a copy of the library in memory. Still another advantage of the DLL approach is that by updating the library you can update all programs that use the library, without having to relink each program.

You can write DOS type programs for an OS/2-based system using the programming tools we described in earlier chapters. For simple protected-mode programs you can use Microsoft's C 5.2 32-bit C compiler and the latest version of Microsoft's Macroassembler (MASM). To develop a 32-bit OS/2 application which utilizes PM, you use Microsoft's OS/2 2.0 Software Developer's Kit which contains all the needed tools.

MICROSOFT WINDOWS 3.0

Microsoft's Windows 3.0 is a relatively inexpensive bridge between the DOS world and the high-powered OS/2 2.0 operating system we discussed briefly in the preceding section. As the name implies, this program uses a graphical user interface (GUI) very similar to Presentation Manager. Windows 3.0 is essentially a very flexible DOS extender which can take advantage of the protected-mode features of the 80286, 386, and 486 processors. It can be operated in any one of three different modes, depending on the processor and memory available in the system. On an 8086/8088-based system, Windows 3.0 must be operated in its real mode. On an 80286-based system with at least 1 Mbyte of extended memory, Windows 3.0 can be run in standard mode, which takes advantage of the protected mode features of the 80286. In real mode and standard mode, only one DOS type task can be run at a time.

On a 386- or 486-based system with at least 2 Mbytes of extended memory, Windows 3.0 can be run in its 386 enhanced mode. In this mode, which is the one we are interested in here, it uses the 386's virtual 8086 mode to run multiple 8086 tasks, and it implements paged virtual memory so it can run programs that require more memory than is physically present in the system.

When you run Windows 3.0 the program manager window shown in Figure 15-30, page 568, appears on the screen. The icons along the bottom of this window

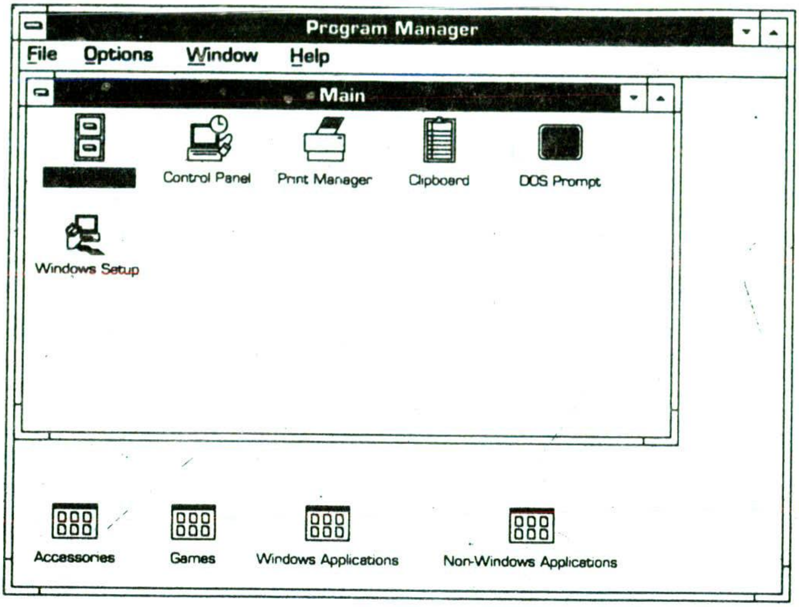


FIGURE 15-30 Microsoft Windows 3.0 Program Manager window display.

represent groups of programs you can execute. If you move the cursor to one of these icons and double-click the left mouse key, another window containing a menu of the programs in that group will appear. You can execute a program in the group by just double-clicking on the name of the program in the menu. If you want to start another program running at the same time, you can "minimize" the window for the running program to put that program in the background and then start another program running. The background program continues running after you start the new program in the foreground. Windows even allows you to specify the percentages of time you want the processor to spend on the background task and on the foreground task. Windows 3.0 keeps a task list of the currently running tasks. You can switch a task from background to foreground by bringing up the task list and clicking on the desired task.

One of the program icons in the program manager window is the file manager. When you double-click on this icon, it opens a file window and shows you a tree of the files and subdirectories in your current directory. In this window you can use the mouse to perform the usual file operations such as copy, delete, rename, etc. The point of all this is that instead of typing in commands, you can perform almost any operation by just moving the cursor to the appropriate location on the screen and clicking the mouse key. Windows 3.0 also has a very versatile on-screen help system which you can pop up as needed.

Windows 3.0 separates programs into two categories, windows applications and nonwindows applications. During setup Windows 3.0 scans your disk drive(s) and puts the programs it finds into the correct category. Programs not specifically written for 3.0 will be classified as nonwindows applications. In the real and standard mode, nonwindows applications are run in full-screen mode similar to the way they would run in a pure DOS environment. Windows applications take advantage of the GUI. Among the windows application-type programs that come with Windows 3.0 are a word processor, notepad, paintbrush, calendar, clock, print spooler, and a card file.

To write a simple Windows 3.0 application program you can use the Asymetrix Corp *Toolbook* which comes with Windows 3.0. Toolbook includes some impressive demonstrations. To develop more complex Windows 3.0 applications, you need tools such as version 2.0 of Borland's C + + or the Microsoft Windows 3.0 Software Development Kit. The programming guide that comes with this tool set contains a sequence of templates that you can use to develop a custom application. We had hoped to rewrite the SDKCOM1 program from Chapter 14 as an example windows application program, but we ran out of space and time. Maybe we can include this in the next book.

THE INTEL 80486 MICROPROCESSOR

The 32-bit 486 is the next evolutionary step up from the 386. The basic processor unit used in the 486 is the same as that used in the 386, so all of the preceding discussion of the 386 applies to the 486. All we have to discuss here are the additions and enhancements that designers were able to add by increasing the number of transistors on the die from 300,000 to about 1,200,000.

As you can see in Figure 15-31, one of the most obvious features included in a 486 is a built-in math

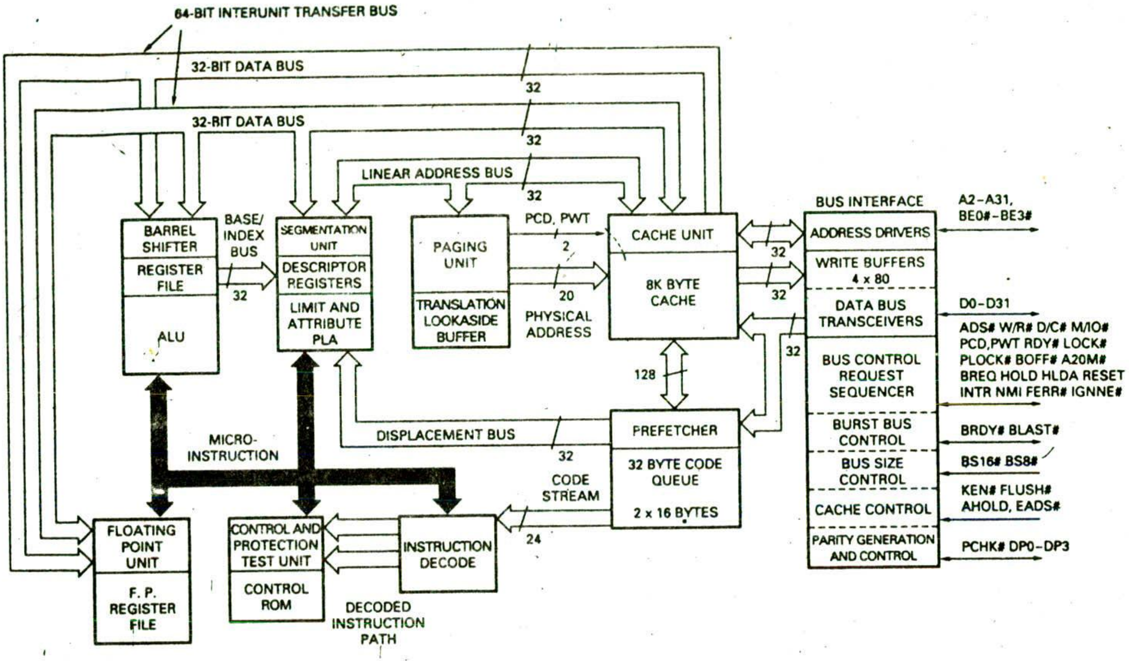


FIGURE.15-31 Intel 486 internal block diagram. *(Courtesy Intel Corporation)*

coprocessor. This coprocessor is essentially the same as the 387 processor used with a 386, but being integrated on the chip allows it to execute math instructions about three times as fast as a 386/387 combination.

Another fairly obvious feature included in a 486 is an 8-Kbyte code and data cache. This four-way set-associative cache works in basically the same manner as the external caches we described in Chapter 11. One difference is that a "line" for this cache is 16 bytes instead of 4 bytes.

A less obvious 486 improvement is the five-stage instruction pipeline scheme that allows it to execute instructions much faster than a 386. This scheme, commonly used in RISC processors, allows several instructions to be "in the pipeline" at a time. The 486 will fetch several instructions ahead of time, and while it is executing one instruction, it will decode and as soon as possible, start the execution of the next instruction. The 486 may actually be executing parts of several instructions at the same time. For example, suppose the 486 is given the meaningless sequence of instructions:

MOV AX, MEMORY_LOCATION
ADD CX, BX
SHR AX, 1
MOV MEMORY_LOCATION, CX

A few clock cycles before it gets to the first of these instructions, the 486 will have prefetched all these simple instructions and started decoding them. As the MOV AX, MEMORY_LOCATION instruction executes, decoding of the ADD instruction will be completed. Since the ADD instruction does not uses the buses or the data read in from memory by the MOV instruction, it can be executed before the MOV AX instruction is complete. Likewise, decoding of the SHR instruction will be completed while the ADD CX, BX instruction is executing, and on the next clock cycle the SHR instruction will be executed. During the clock cycle that the SHR instruction executes, the decoding of the MOV MEMORY_LOCATION, CX instruction will be completed, and the address of the memory location will be output on the address bus. On the next clock cycle the word in AX will be output on the data bus. This extensive overlapping of operations makes it possible for the 486 to execute many of its commonly used instructions in, effectively, a single clock cycle. The fetching, decoding, and executing of each of these instructions actually takes several clock cycles, but since these operations are overlapped with the decoding and execution of other instructions, the net time for each of the instructions is only one clock cycle. As an example of this, a 16-bit memory-write operation that takes 22 clock cycles to execute on an 8088 and 4 clock cyles to execute on a 386 takes only 1 clock cycle to execute on a 486. The conditional jump instructions have also benefited greatly from the pipelining in the 486. When the 486 decodes a conditional jump instruction, it automatically prefetches one or more instructions from the jump destination address just in case the jump is taken. If the branch is taken, then the 486 does not have to wait through a bus cycle for the first instruction at the branch address. A conditional jump instruction which takes 16/4 clock cycles on an 8088 and 8/3 clock cycles on a 386 takes only 3/1 clock cycles on a 486.

Most of the other improvements included in the 486

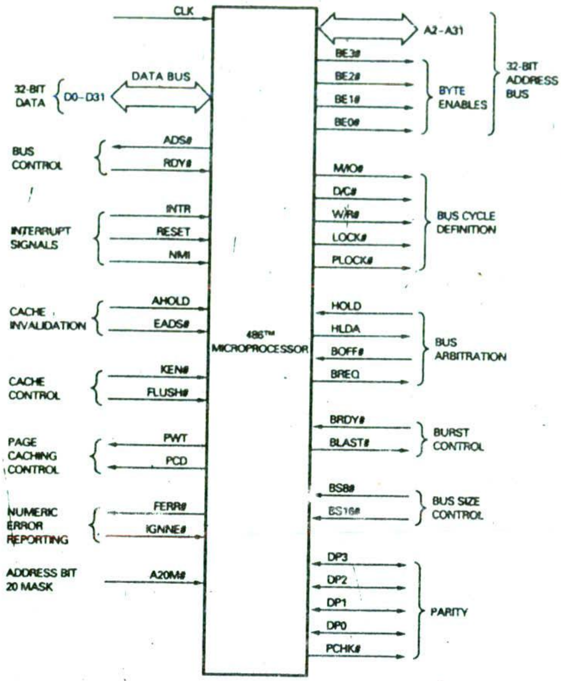


FIGURE 15-32 486 functional signal groups. (*Courtesy Intel Corporation*)

involve hardware signals and interfacing. To make room for the additional signals, the 486 is packaged in a 168-pin pin grid array package instead of the 132-pin PGA used for the 386. Figure 15-32 shows the 486 signals in functional groups. We will briefly work our way through some of these to give you an overview of the major new features.

The 486 data bus, address bus, byte enable, ADS#, RDY#, INTR, RESET, NMI, M/IO#, D/C#, W/R#, LOCK#, HOLD, HLDA, and BS16# signals function as we described for the 386, so these hold no surprises. The 486 requires a 1 × clock instead of 2 × clock required by the 386.

A new signal group on the 486 is the parity group, DP0–DP3, and PCHK#. These signals allow the 486 to implement parity detection/generation for memory reads and writes. During a memory write operation, the 486 generates an even parity bit for each byte and outputs these bits on the DP0–DP3 lines. As we described for the IBM PC in Chapter 11, these bits will be stored in a separate parity memory bank. During a read operation the stored parity bits will be read from the parity memory and applied to the DP0–DP3 pins. The 486 checks the parities of the data bytes read and compares them with the DP0–DP3 signals. If a parity error is found, the 486 asserts the PCHK# signal.

Another new signal group consists of the burst ready signal, BRDY#, and the burst last signal, BLAST#. These signals are used to control burst-mode memory reads and writes. Here's how this works. A normal 486 memory-read operation to, for example, read a line into the cache requires 2 clock cycles. However, if a series of reads is being done from successive memory locations, the reads can be done in burst mode with only 1 clock cycle per read. To start the process the 486 sends out the first address and asserts the BLAST# signal high. When the external DRAM controller has the first data word ready on the data bus, it asserts the BRDY# signal. The 486 reads the data word and outputs the next address. Since the data words are at successive addresses, only the lower address bits need to be changed. If the DRAM controller is operating in the page or the static column modes we described in Chapter 11, then it will only have to output a new column address to the DRAM. In this mode the DRAM will be able to output the new data word within 1 clock cycle. (If the DRAM is not fast enough for a high-speed 486, then two DRAM banks can be interleaved to gain the required speed.) When the processor has read the required number of data words, it asserts the BLAST# signal low to terminate the burst mode.

The final signals we want to discuss here are the bus request output signal, BREQ; the back-off input signal, BOFF#; the HOLD signal; and the hold-acknowledge signal, HLDA. These signals are used to control sharing the local 486 bus by multiple processors (bus masters). When a master on the bus needs to use the bus, it asserts its BREQ signal. An external priority circuit will evaluate requests to use the bus and grant bus use to the highest-priority master. To ask the 486 to release the bus, the bus controller asserts the 486 HOLD input or BOFF# input. If the HOLD input is asserted, the 486 will finish the current bus cycle, float its buses, and assert the HLDA signal. To prevent another master from taking over the bus during a critical operation, the 486 can assert its LOCK# or PLOCK# signal.

Because of all the possible variations, we don't have space here to discuss the operation of the 486 cache control signals. Consult the 486 data sheet if you need to know about these and the few other signals we didn't get around to.

The 486 has six additional instructions beyond those of the 386. INVD and WBINVD invalidate the cache, and INVLPG invalidates a TLB entry. The BSWAP instruction swaps the order of the bytes in a 32-bit register. This is useful in interfacing with, for example, an IBM mainframe which stores the least significant byte in the upper bits of a data word. The XADD and CMPXCHG instructions are used to work with semaphores such as those we showed you in Figure 15-3.

NEW DIRECTIONS

Microprocessor and microcomputer evolution has been proceeding very rapidly in the last few years, and the rate of evolution seems to be increasing. As we have shown in the preceding chapters, the overall direction of this evolution is toward microcomputers with greater screen resolution, more memory capability, larger data words, higher processing speeds, and network communication. David House of Intel recently revealed Intel's current plans for evolution beyond the 486. The 586, expected in 1992, will contain about 2 million transistors, and the 686, expected in 1996, will contain about 5 million transistors. The added transistors, of course, will allow larger caches and many new functions to be implemented on the chips. The 786, to be available some time in the late 1990s, is projected to contain a 2-Mbyte cache, six separate integer and floating-point processors, and a complete digital video interactive or similar user interface. The 786 will maintain compatibility with the 386/486 instruction set and operate with a 250-MHz clock.

Throughout this book we have discussed the operation and evolution of primarily one processor family, the Intel 8086 family. We did this so that we could develop some depth rather than just an overview of all the different processors. To finish the book, however, we want to briefly discuss some other types of microcomputer systems that you should be aware of.

RISC Machines

High-performance engineering workstations often use a *reduced instruction set computer* or RISC-type processor. The term RISC is not precisely defined, but some of the main characteristics often associated with a RISC processor are the following:

1. The instruction set is limited to simple arithmetic-, logic-, load-, and store-type instructions. Fewer instructions and limited addressing modes mean a simpler and faster instruction decoder.
2. Extensive pipelining is used to achieve one-clock-cycle instruction execution. The 486 is a CISC processor, but as we described in a previous section, it uses a four-stage pipeline to achieve one-clock-cycle execution for many instructions. The assembler/compilers used for developing RISC programs are designed to put instructions in an order which will keep the pipeline full as much of the time as possible. To further overlap operations, some RISC machines use a *Harvard architecture*, which has separate data buses for code fetches and for data read/write operations.
3. Execution of conditional jump instructions is delayed to allow time to load the pipeline with instructions from the jump destination, or instructions from the jump destination are fetched ahead of time, as we described for the 486 in a previous section.
4. The CPU contains a large number of on-chip registers to give improved access to data operands.

One example of a current RISC implementation standard is the Scalable Processor Architecture RISC computer (SPARC) developed by Sun Microsystems, and implemented in their SPARC stations. Fujitsu and Cypress Semiconductor have produced chip sets for this standard. Other common RISC chip sets are the Motorola 88000, the MIPS R3000, and the AMD 29000. A single-chip RISC processor now available is the Intel i860™.

The 1.2 million–transistor i860 contains a 64-bit RISC-based core with Harvard architecture, a floating-point coprocessor, and a graphics coprocessor with 3-D graphics capability. The processors in the i860™ operate relatively independently of each other, so they can all be working in parallel. With a 40-MHz clock an i860™, can perform at peak rate of 80 million floating-point operations per second (MFLOPS), or 85,000 drystones. (The drystone rating represents the relative performance of a computer executing a standard "benchmark" program.) Incidentally, the i860 does not use segmentation, but it does allow 386-type virtual memory paging. The data sheets for this device are a good source of information about RISC implementation.

Parallel Processing

Some computer applications such as analyzing weather data, simulating aircraft designs, or creating the graphics for high-tech science fiction movies require massive amounts of computing. The microcomputers we have discussed in this book so far do not operate nearly fast enough to be practical for many of these applications, so *supercomputers* are used. The peak execution speeds of the fastest current supercomputers are in the range of a few gigaFLOPS.

Most supercomputers are built by connecting several processing elements in parallel. One connection scheme, commonly called a *farm*, allows multiple processors to access a single large memory with a common bus. The difficulty with a simple bus structure such as this is that processors compete for shared resources. If one processor is using the bus, others must wait. This slows down the overall processing speed.

One of the more efficient multiprocessor architectures is the hypercube topology developed by Seitz and Fox at Caltech. A diagram of this topology is shown in Figure 15-33. Each node in the system consists of a complete processing unit which has the ability to communicate with other units. Each processor unit is typically connected to communicate with its nearest neighbors as

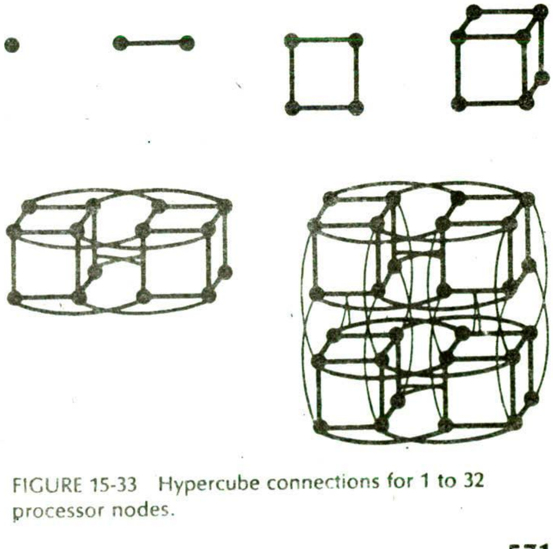


FIGURE 15-33 Hypercube connections for 1 to 32 processor nodes.

shown. The number of nodes can be expanded to give the power and speed needed to handle the problem the computer is being used to solve.

Intel Supercomputing Systems Division (ISSD) has produced the iPSC family of commercial products based on the hypercube topology. One of these, the iPSC/i860™ can be configured with up to 128 nodes, each containing a complete i860™-based microcomputer with high-speed network-type communication capability. The advantage of this structure is that each processor has enough memory to operate relatively independently, and communication between processors can take any one of several routes, instead of being limited to a single bus. Peak execution rates for the iPSC/i860™ range from 480 MFLOPS to 7.5 gigaFLOPS, depending on the number of processing units. These rates compare favorably with Cray Research Y-MP supercomputer's maximum execution rate of about 8 gigaFLOPS. However, because the iPSC/i860™ uses a larger number of common LSI components instead of a single or small number of very high speed processors made with gallium arsenide technology, the cost is less.

To program parallel computers such as these, new programming languages have had to be developed. A couple of these that you may be hearing more about are Scientific Computing Associates' C-Linda language and AIL Ltd.'s STRAND 88 language.

Expert Systems, Neural Networks, and Fuzzy Logic

INTRODUCTION

Artificial Intelligence or AI is the general term used to describe computers or computer programs which solve problems with "intuitive" or "best-guess" methods often used by humans instead of the strictly quantitative methods usually used by computers. Expert systems, neural networks, and fuzzy logic are the most common types of AI currently in use. These three are in relatively early stages of development and implementation, but they all have extensive implications for our lives.

EXPERT SYSTEMS

Probably the most developed area of AI at present is the area of *expert systems*. An expert-system program consists of a large data base and a set of rules for searching the data base to find the best solution for a particular type of problem. The data base and rules are developed by questioning "experts" in that particular problem area. The data base for a medical diagnosis expert system, for example, is built up by extensive questioning of experts in each medical specialty.

Unlike most computer programs, which require complete information to make a decision, expert system programs are designed to make a best guess, based on the available data, just as a human expert would do. A medical diagnosis expert system, for example, will indicate the illness that most likely corresponds to a given set of symptoms and test data. To enable it to make a better guess, the system may suggest additional tests to perform.

One advantage of a system such as this is that it can make the knowledge of many experts readily available to a physician anywhere in the world via a modem connection. Another advantage is that the data base and set of rules can be easily updated as new research results and drugs become available. Other examples of expert system programs are those used to lay out PC boards and those used to lay out ICs.

NEURAL NETWORKS

Programs for some problems such as image recognition, speech recognition, weather forecasting, and three-dimensional modeling are not easily or accurately implemented on fixed-instruction-set computers such as 386/i486-based systems. For applications such as these, a new computer architecture, modeled after the human brain, shows considerable promise.

As you may remember from a general science class, the brain is composed of billions of neurons. The output of each neuron is connected to the inputs of several thousand other neurons by synapses. If the sum of the signals on the inputs of a neuron is greater than a certain threshold value, the neuron "fires" and sends a signal to other neurons. The simple op-amp circuit in Figure 15-34*a* may help you see how a neuron works. Let's assume the output of the comparator is initially low. If the sum of the input signals to the adder produces an output voltage more negative than the comparator threshold voltage, the output of the comparator will go high. This is analogous to the neuron firing. The weight or relative influence of an input is determined by the value of the resistor on that input. Figure 15-34*b* shows a symbol commonly used to represent a neuron in neural network literature and Figure 15-34*c* shows a simple mathematical model of a neuron.

As with the neurons in the human brain, the neurons in a neural network are connected to many other neurons. Figure 15-34*d* shows a simple three-layer neural network. This network configuration is referred to as "feedforward," because none of the output signals are connected back to the inputs. In a "feedback" or "resonance" configured network, some intermediate or final output signals are connected back to network inputs. Researchers are currently experimenting with many different network configurations to determine the one that works best for each type of application.

Neural network based computing can be implemented in several ways. One way is to use a dedicated processor

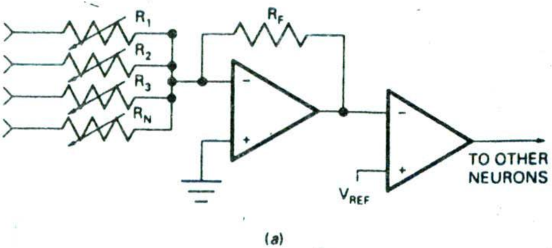
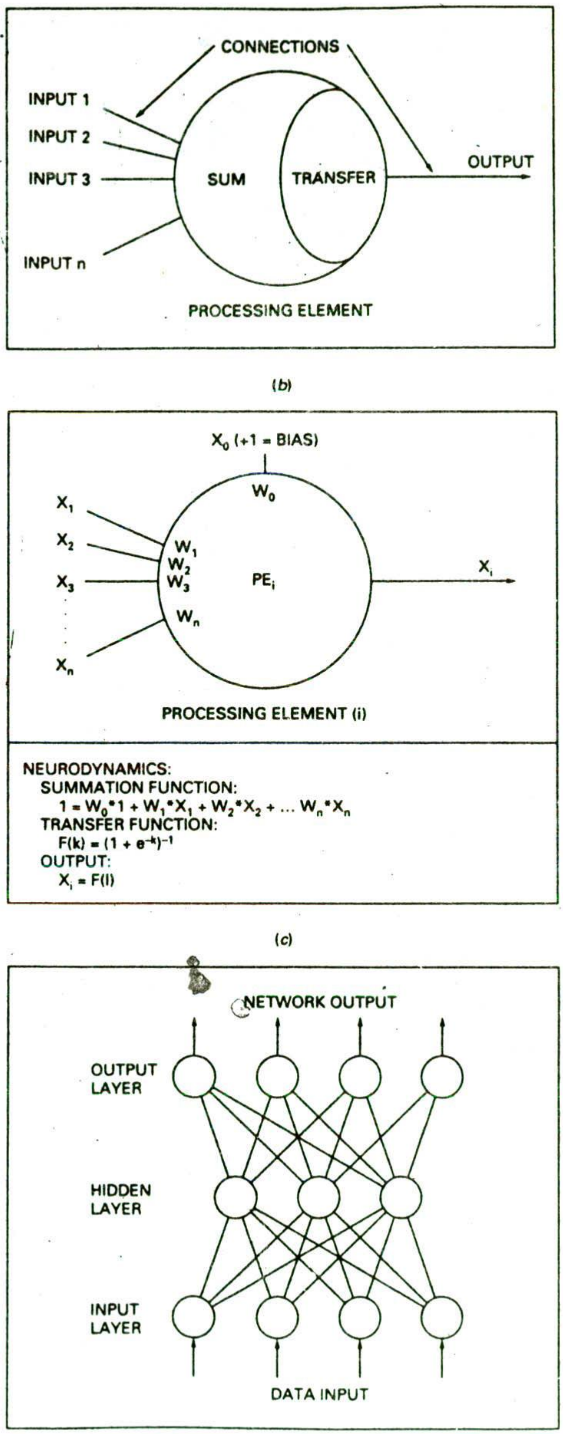


FIGURE 15-34 (a) Op-amp model of a neuron. (*See also next page.*)



(b)

NEURODYNAMICS:
SUMMATION FUNCTION:
$1 = W_0{*}1 + W_1{*}X_1 + W_2{*}X_2 + \ldots W_n{*}X_n$
TRANSFER FUNCTION:
$F(k) = (1 + e^{-k})^{-1}$
OUTPUT:
$X_i = F(I)$

(c)

(d)

FIGURE 15-34 (*Continued*) (*b*) Neural network processing element. (c) Mathematical representation of processing element. (*d*) Simple three-layer neural network. (*Courtesy NeuralWare, Inc.*)

for each neuron. The large number of neurons usually makes this impractical, and most applications don't need the speed capability. An alternative approach is to use a single processor and simulate neurons with lookup tables. The lookup table for each neuron contains the connections, input weight values, and output equation constants. Hecht-Nielsen Neurocomputers markets a PC/AT-compatible coprocessor board which uses this approach.

A neural network can also be implemented totally in software. NeuralWare, Inc. markets neural net simulation programs for both PC and Macintosh type computers. These packages can be used to learn about neural nets or develop actual applications which do not have to operate in real time. Another interesting neural network program is BrainMaker from California Scientific Software.

Neural network computers are not programmed in the way that digital computers are, they are trained. Instead of being programmed with a set of rules the way a classic expert system is, a neural network computer learns the desired behavior. The learning process may be supervised, unsupervised, or self-supervised.

In the supervised method a set of input conditions and the expected output conditions are applied to the network. The network learns by adjusting or "adapting" the weights of the interconnections until the output is correct. Another input-output set is then applied, and the network is allowed to learn this set. After a few sets the network will have learned or generalized its response so that it can give the correct response to most applied input data.

The scheme used to adapt the network is called the learning rule. As an example, one of the simplest learning rules that can be used is the Hebbean Learning Law. This law decrees that each time the input of a neuron contributes to the firing, its weight should be increased, and each time an input does not contribute, its weight should be decreased. This is somewhat analogous to a positive-negative reinforcement scheme often used in human behavior modification. In the case of the network the result is that these successive "nudges" adapt the network output to the desired result.

The major advantages of neural networks are these:

1. They do not need to be programmed; they can simply be taught the desired response. This eliminates most of the cost of programming.

2. They can improve their response by learning. A neural network designed to evaluate loan applications, for example, can automatically adapt its criteria based on loan-failure feedback data.

3. Input data does not have to be precise, because the network works with the sum of the inputs. A neural network image-recognition system, for example, can recognize a person even though he or she has a somewhat different hairstyle than when the "learning" image was taken. Likewise, a neural network-based speech-recognition system can recognize words spoken by different people. Traditional digital techniques have a very hard time with these tasks.

4. Information is not stored in a specific memory location the way it is in a normal digital computer; it is stored *associatively* as a network of interconnections and weightings. The result of this is that the "death" of a few neurons will usually not seriously degrade the operation of the system. This characteristic is also fortunate for us humans!

Software-based neural networks can be used for nonrealtime applications such as forecasting the weather or the stock market. For realtime applications such as image recognition and speech recognition, the software methods are obviously not fast enough. University researchers and companies such as TRW and Texas Instruments are working on ICs, which implement neural networks in hardware. In the not-too-distant future these ICs should allow you to talk to your computer instead of using a mouse, allow your computer to read typed messages to you, and allow your car to drive itself down the freeway.

FUZZY LOGIC

Consumer products such as video camcorders, cameras, refrigerators, washing machines, and automobiles are increasingly using fuzzy logic control circuits. Linking the term fuzzy, which here means "not precisely defined," with the term logic may seem to create an oxymoron like "work party," but the concept is very real. The original work on fuzzy logic was done by Professor Lofti A. Zadeh at U.C. Berkeley in the mid-1960s, but Japanese companies have been the main ones to patent the technology and implement it in products.

A fuzzy logic controller is programmed with rules as is an expert system, but the rules are very flexible. Figure 15-35 shows the graphic method Professor Bart Kosko of the University of Southern California uses to illustrate the difference between traditional fixed value logic and fuzzy logic. Each corner of the cube represents one of the eight possibilities for a three-variable digital logic function. For this example, let's assume that the function is true for the 010, 001, and 100 combinations shown. In a traditional digital logic system the variables can only have values of 0 or 1, so the only values that will produce a true output are these three. In a fuzzy logic system the variables can have values other than 1 or 0, so the set of all the possible values that will produce a true output is represented by the triangular plane formed by the three points. One way to look at this is that traditional digital logic is just a special case of fuzzy logic.

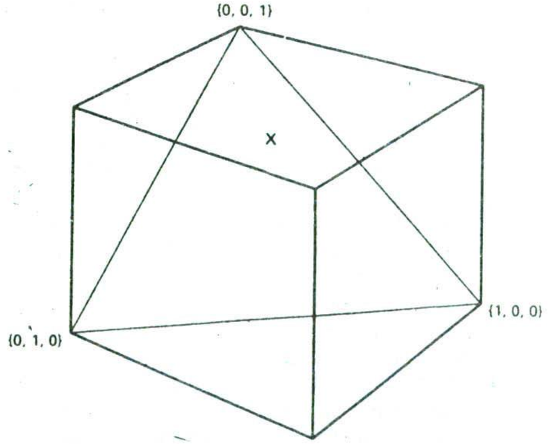


FIGURE 15-35 Comparison of binary logic values and fuzzy logic values for a 3-input function.

One advantage of fuzzy logic systems is that they can work with imprecise terms such as cold, warm, hot, or near boiling that humans commonly use. In hardware terms this means that a fuzzy logic system often doesn't need precise A/D converters. The Sanyo Fisher Corp. Model FVC-880 camcorder, for example, uses fuzzy logic to directly process the outputs from six sensors and set the camera lens for best possible focusing and exposure.

Fuzzy logic can provide very smooth control of mechanical systems. The fuzzy logic-controlled subway in Sendai, Japan, is reportedly so smooth in operation that standing riders do not use the hand straps during starts and stops.

In the United States, Togai InfraLogic, Inc. in Irvine, California, has developed a Digital Fuzzy Processor chip. They have also developed a Fuzzy-C compiler which can be used to write a program containing the rules and knowledge base for the processor.

SUMMARY

The three AI approaches we have discussed in this section will obviously not replace standard digital computers for most applications, especially those that involve numerical processing, but they do give some new choices for difficult applications. The most likely scenario for the future is that a combination of these techniques will be used to design a system which best fits the particular application. The results should be very exciting.

EPILOGUE

This book has been able to show you only a small view of current microcomputers and the directions in which they seem to be evolving. Hopefully we have given you enough of a start that you can continue learning on your own and play a part in the evolution. Whenever you feel overwhelmed by the amount of new material there is to learn, remember the 5-minute rule and the old saying "Grapevines and people bear the best fruit on new growth."

CHECKLIST OF IMPORTANT TERMS AND CONCEPTS IN THIS CHAPTER

If you do not remember any of the terms in the following list, use the index to help you find them in the chapter for review.

Multiuser, multitasking

TSR program

Time-slice and preemptive priority-based scheduling

Semaphore

Deadlock

Critical region

Overlay

Bank switching

Expanded memory

Extended memory

Descriptor table

Virtual memory

Memory-Management Unit

80286
- Real address mode
- Protected virtual address mode
- Interrupt, exception, fault, trap

80386
- Architecture, pins, and signals
- System connections and interface buses
  - ISA bus standard
  - EISA bus standard
  - MicroChannel Architecture Bus
- Real-mode operation
- Protected-mode operation
  - Segmentation and virtual memory
  - Segment privilege levels and protection
  - Call gate
  - I/O privilege levels
  - Interrupt and exception handling
  - Task switching and task state segment
  - Paging mode
  - Flat system memory model
- Virtual 8086 mode operation
- Virtual machine monitor
- Index scaling
- OS/2
- Microsoft windows

80486
- Pipelining
- Cache
- Floating-point processor

Parallel processing
- Hypercube topology

RISC machine

Artifical intelligence

Expert system

Neural network

Fuzzy logic

5-minute rule

Grapevines and new growth

REVIEW QUESTIONS AND PROBLEMS

1. *a.* Describe the basic operation of a TSR program and draw a memory map to show how a TSR program is loaded in a DOS based system.
   b. Use a diagram to help explain how a passive TSR gets executed after it is installed.

2. Briefly describe the two types of scheduling commonly used in multiuser/multitasking operating systems.

3. Suppose that two users in a time-share computer system each want to print out a file. How can the system be prevented from printing lines from one file between lines of the other file?

4. Define the term deadlock and describe one way it can be prevented.

5. Define the term critical region and show with 8086 assembly language instructions how a semaphore can be used to protect a critical region.

6. Describe how an overlay scheme is used to run programs such as compilers which are too large to be loaded into physical memory all at once.

7. *a.* Describe how bank switching is implemented in a microcomputer system.
   b. Describe how LIM 4.0-type expanded memory works in a DOS-based system.
   c. How is extended memory different from expanded memory?

8. *a.* Define the term virtual memory and use Figure 15-8 to help you briefly describe how a logical address is converted to a physical address by a memory management unit.
   b. What action will the MMU take if it finds that a requested segment is not present in physical memory?
   c. What is another major advantage of the indirect addressing provided by descriptor tables, besides the ability to address a large amount of virtual memory?

9. List the four major processing units in an 80286 microprocessor and briefly describe the function of each.

10. Describe how the real-mode operation of an 80286 is different from protected-mode operation.

11. Define the terms interrupt, exception, fault, and trap.

12. Explain how an 80286 is switched from real address mode to protected virtual address mode and how it is switched back to real address mode operation.

13. *a.* Show the computations which tell how much virtual memory an 80286 can address.
b. What factors determine how much physical memory an 80286 can address?

14. *a.* List three major advances that the 80386 microprocessor has over the 80286.
b. What is the main difference between the 386DX processor and the 386SX processor?
c. What is the purpose of the 386DX BE0-BE3 signals?

15. *a.* How is the EISA bus different from the ISA bus?
b. If you found an interface board lying on the bench, what is one way you could tell whether it came from an EISA-based system or from a MicroChannel Architecture system?
c. Briefly compare the EISA and MCA methods of arbitrating bus requests from multiple masters or DMA slaves.

16. *a.* Show the computations which tell how much virtual memory a 386 can address.
b. How much physical memory can a 386 address in real mode and in protected mode?

17. *a.* Give the names of the two parts of a 386 protected-mode address.
b. Using Figure 15-21 to help, describe how a 32-bit virtual address for a data segment location in a task's local memory is translated to the actual 32-bit physical address for a 386 operating in segments only protected mode.
c. How would the discussion in part b differ if the desired memory location were in the global memory area?
d. How does a 386 keep track of where the global descriptor table and the currently used local descriptor table are located in memory?
e. Why is the length of the segment included in the descriptor for a segment?

18. How are tasks in a 386 system protected from each other?

19. How can operating system kernel procedures and data be protected from access by application programs in a 386 system?

20. In a 386 system a task operating at a level 2 privilege can in a special way call a procedure at a higher privilege level. Describe briefly the mechanism that is used to make this access.

21. *a.* A 386 maintains a task state segment for each active task in a system. How are these task state segments accessed?
b. Briefly describe how a 386 does a task switch using a FAR JMP or a FAR CALL instruction.

22. *a.* Use Figure 15-26 to help you explain how a 386 computes a physical address when its paging mode is enabled.
b. What is the advantage of paged-based virtual memory over segments only based virtual memory?
c. Define the term simple flat memory model and the term paged flat memory model for a 386.

23. *a.* How is a 386 switched into virtual 8086 mode during a task switch?
b. Briefly describe the response of the virtual machine monitor when a real-mode 8086 program executes an INT 21H instruction.

24. Using the program in Figure 15-29*a* as a model, write a program which uses some of the new 386 instruction features to treat the four words in table as a 64-bit word, and rotate it 8 bits to the left.

25. Describe three major additions or improvements that the 486 processor has over the 386 processor.

26. List three major features characteristic of a RISC-based computer and describe how each of these features helps produce faster execution.

27. What are the major advantages of using parallel processors with, for example, a hypercube connection architecture over using a single fast processor?

28. *a.* Describe the basic operation of a neuron in a neural computer.
b. How is the "programming" of a neural network computer different from the programming of a standard, fixed-instruction-set computer?
c. List some advantages of a neural network–type computer.
d. For what types of applications are neural network–type computers best suited?

29. *a.* How is a fuzzy logic control system different from a traditional digital logic control system?
b. What are some advantages of fuzzy logic?