2

An Overview of the Machine and Run-Time Environment

2.1 The Machine

Operating systems deal with the details of devices, processors, and memory – they cannot be designed without some notion of a machine. This book uses the IBM PC or its alternatives, the PC/XT and PC/AT microcomputer systems. We refer to any of them generically as a ‘‘PC.’’ The PC was chosen because of its popularity and low cost. The 8088/8086/80286 microprocessers used in PCs are sufficiently simple to serve as models of a general purpose processor and sufficiently complex to illustrate the details of operating system software.

The remainder of this chapter introduces the PC hardware. From the operating system designer’s point of view, the firmware that the vendor supplies with a PC in ROM† is part of the hardware. From the vendor’s point of view, having services built into ROM, instead of the underlying hardware, makes it possible to have multiple models of PCs that use slightly different underlying hardware but which all run programs in the same way. Therefore, this chapter describes the services supplied by ROM firmware as well as pertinent features of the processor, memory, and communication devices. It explains the PC architecture, including processor mechanisms like segment addressing, the stack, and vectored interrupts, as well as peripheral devices. The discussion focuses on the Intel 8088, the microprocessor used on the original PC. The 8086 and 80286 processors that appeared on later models have essentially the same register configuration instruction set as the 8088. Although the exact details of the hardware are not important, the basic ideas are.

---

† ROM stands for Read Only Memory. It contains non-volatile code and data which may be read by the processor but may not be modified.

2.2 Physical Organization Of The PC

A PC is constructed from a set of printed circuit boards. One printed circuit board is the *motherboard* containing the microprocessor, computer memory and associated circuits. In addition, there is a collection of sockets into which other printed circuit *cards* may be inserted. These sockets form the *backplane* and serve two functions: they hold the cards in place and provide electrical connections to them. These wired-together sockets are collectively called a *bus*. Some PCs do not have a motherboard; in such systems, the microprocessor and memory circuits are put on a card inserted into the backplane.

Figure 2.1 A motherboard with backplane and cards.

Cards in the backplane are usually used for additional memory or for I/O devices. Instead of one specific set of cards, vendors offer a variety of cards so that each customer can plug together a customized configuration. For example, one type of board contains additional memory and a real-time clock device. Another contains less memory but includes devices that communicate with modems or printers. Yet another contains a high-resolution graphics interface.

A board can communicate with other boards only by passing signals across the bus. When the processor needs to write to a memory card on the bus, it places the address and data on the bus for the memory card to retrieve and store. When it needs to read data or fetch instructions, it places the address on the bus and asks the memory card to supply the data value. Naturally, the details of the bus design specify exactly how the cards manipulate and respond to signals to make these operations work correctly; for our purposes, such details are unimportant.

Independent of the physical arrangement and number of cards, memory must always be logically contiguous. To permit memory from several boards to be mapped into a contiguous address range, each board contains switches that can be changed. Thus, it is possible to configure two identical memory boards so that one responds to low memory addresses while the other responds to higher addresses. Likewise, the I/O device interfaces contain switches, so two boards can be configured to represent two distinct devices even though they happen to be the same physical type.

2.3 Logical Organization Of The PC

The operating system is concerned with the logical organization of the machine, not its physical organization. The next sections describe the highlights of the PC hardware that affect some of the code described later. It is not important to understand all the details now. Programmers familiar with the PC architecture can skip most of the material; other readers should skim through the text to learn the basic ideas and refer to it when specific details arise in later chapters.

2.3.1 The Address Space

Memory on the 8088 is divided into 8-bit quantities called *bytes*, with the byte being the smallest addressable unit. The term *character* is often used in place of *byte* because bytes are commonly used to hold characters. Most instructions can operate on either bytes or *words* (16 bits). Word operations may refer to even or odd byte addresses; the operation always affects the addressed byte and the next higher byte. Programmers unfamiliar with the 8088 should note that the address of the low-order byte of a word is the same address as that of the word itself.

The 8088 can address up to 1024K bytes of memory (K=1024) because physical addresses are 20 bits long. However, on the PC, addresses larger than 640K are reserved for the console display and ROM software.

While a physical address for the 8088 is specified by a single 20-bit unsigned number, the instruction set of the processor cannot handle 20-bit addresses. Instead, the 8088 uses logical addresses specified by two unsigned 16-bit numbers, one called the *segment* and the other the *offset*. A logical address expressed in this way is called a *segment:offset* address and is written in the form

segment : offset.

We will always express processor addresses, both 20-bit physical address and segment : offset address, in hexadecimal (base 16) notation.

A segment : offset address is translated into a 20-bit physical address (5 hexadecimal digits) by multiplying the segment component by 16 (decimal) and adding the offset. In hexadecimal notation, multiplying by 16 is equivalent to appending a low-order zero. For example, the segment : offset address

14CF:B35E

is translated into the physical address

```
 14CF0
+B35E
------
 2004E
```

Also observe that a given physical address can be represented in more than one way using segment : offset addresses. The above address, for example, can also be represented as the segment : offset address

13CF:C35E.

Segment : offset addresses require four bytes of storage, two bytes for the segment and two for the offset. The offset component is stored in the two lower-address bytes and the segment component is stored in the two higher-address bytes.

The first 1K of 8088 memory, from 00000 to 003FF, is reserved for interrupt vectors. An *interrupt vector* is a four-byte segment : offset address of code to be executed upon receipt of a hardware or software interrupt. There is room for 256 interrupt vectors in this area. Interrupts will be discussed later in this chapter.

Addresses 00400 through 9FFFF constitute available user memory. Addresses A0000 through FFFFF are reserved for video graphics and ROM subroutines. All addresses refer to one physical address space; there is no memory management hardware on the basic PC. (While the 80286 does support a form of memory management, this feature will not be discussed further.)

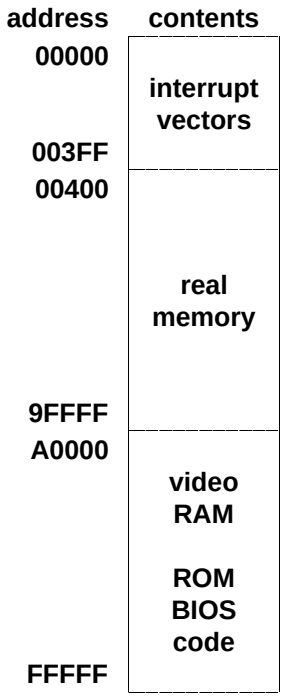

Figure 2.2 The address space of the PC microcomputer system.

2.3.2 Registers In The 8088

Figure 2.3 illustrates that the 8088 processor contains twelve 16-bit registers, an Instruction Pointer, and a 16-bit FLAGS register.

Register	Use
AX	general purpose
BX	general purpose
CX	general purpose
DX	general purpose
SP	stack pointer
BP	base pointer
SI	source index
DI	destination index
CS	code segment
DS	data segment
ES	extra segment
SS	stack segment
IP	instruction pointer
FLAGS	flags word

Figure 2.3 Registers in the 8088

The *segment* registers CS, DS, ES and SS are used as default segment addresses for most instruction and data references using the other registers. For example, CS:IP provides the segment : offset address of the next instruction to execute. Most references to data use DS as the default segment component of their segment : offset addresses. Stack operations use segment : offset addresses of the form SS:SP. The C compiler uses SS:BP to point to the stack frame on the run-time stack for the currently active procedure.

The PC-Xinu system uses separate, fixed code and data segments using segment registers CS and DS, respectively; the stack segment SS coincides with the data segment DS. This provides 64K of code space in the code segment and 64K for data and stack space in the data segment.

The low- and high-order bytes of the registers AX, BX, CX and DX can be accessed directly using L and H to replace the X in the register designation. Thus AL refers to the low-order byte of the AX register, and AH refers to the high-order byte.

2.3.3 FLAGS Word

The FLAGS word contains the *interrupt flag* and condition code bits. The interrupt flag, which is bit 9 of the FLAGS word, is the only component which concerns us here. Figure 2.4 shows the layout of the FLAGS word.

Bits:	15 through 10	9	8 through 0
Contents:	. . .	interrupt flag	. . .

Figure 2.4 The bits of the 8088 FLAGS word.

When the interrupt flag is set (1), the processor will acknowledge hardware interrupts such as those produced by the keyboard or disk drive When the flag is clear (0), the processor will ignore hardware interrupts. Notice that software interrupts will always be acknowledged, independent of the setting of the interrupt flag.

The interrupt flag may be set using the STI (SeT Interrupt flag) instruction and cleared using the CLI (CLear Interrupt flag) instruction.

The FLAGS word contains additional bits which are used for conditional branches. These bits are modified by arithmetic and logical instructions. The two flags that PC-Xinu uses are the *carry flag* and the *zero flag*. These flags are described in Figure 2.5.

Flag	Meaning
CF	Carry Flag set (1) on unsigned overflow/underflow reset (0) otherwise
ZF	Zero Flag set (1) when the result is zero reset (0) otherwise

Figure 2.5 Arithmetic and logical flags

2.3.4 Vectored Interrupts

The 8088 processor employs the conventional *vectored interrupt* scheme for handling exceptions and interrupts from external devices. Whenever an external device needs to communicate with the processor, the device places a signal on the interrupt bus line. If the processor is running with interrupts enabled, it checks the interrupt line after executing each instruction to see whether an interrupt needs processing. To handle the interrupt, the processor disables further interrupts and sends an acknowledgement over the bus, requesting that the interrupting device return an *interrupt type*, which is a one-byte quantity. The device with a pending request receives the acknowledgement and responds by returning its interrupt type, *v*, to the CPU on the data bus. When it receives the interrupt type, the processor pushes the current FLAGS word and the CS:IP address on the stack, loads the CS:IP address from the two words in memory starting at location 4*v*, and continues executing instructions beginning at the new location. The code at this location is called an *interrupt service routine*, or *ISR*.

Each device is assigned a unique interrupt type (and therefore an interrupt vector address), enabling the system software to distinguish among them. Thus, there is no need to poll devices to find out which one needs service when an interrupt occurs because the software can identify devices based on their interrupt types.

It is the programmer's responsibility to insure that an appropriate ISR has been installed before the interrupt occurs so that the interrupt vector location in memory contains the segment : offset address of the ISR. When an interrupt does occur, the stack SS:SP must point to a valid stack address that can hold at least three 16-bit words to save the FLAGS word and the CS:IP address.

An interrupt acts like a procedure call (except that the hardware forces it to occur ''between'' the execution of two instructions in the user's code). The processor executes code in the ISR, eventually returning to the place at which the user's program was interrupted. To make the interrupt transparent to a running program, the ISR must save and restore the state of the machine. On the 8088, saving the state consists of saving any registers used in the ISR. The FLAGS word does not need to be saved because the interrupt mechanism pushes the FLAGS word on the stack at interrupt time. In practice, registers that need to be saved are pushed on the stack. The stack pointer (SP) need not be saved provided that the ISR pops off whatever it pushes on the stack before returning (i.e., restores the stack to its original position).

To prevent the ISR from itself being interrupted by another device, the processor disables interrupts prior to executing code in the ISR. In some cases it may be possible or even desirable to allow for interrupts to be acknowledged during execution of an ISR. In this case, the routine itself may simply enable interrupts.

Interrupt processing ends when the processor executes a *return from interrupt* instruction (IRET). In a single step, the IRET instruction restores the CS:IP segment address and FLAGS word from the stack and returns the stack pointer to its original value. This action reverses that taken by the processor when it detected an interrupt and allows processing to continue where it was interrupted.

2.3.5 Exceptional Conditions

Exceptional conditions are handled by the 8088 exactly like interrupts. An exception, like division by zero, a memory error, or power failure, can be thought of as a hardware detected error. When an exception occurs, the processor pushes the current FLAGS register and CS:IP address onto the stack and loads a new CS:IP from a vector location in memory. As with interrupt vectors, the programmer assumes responsibility for storing a valid address of an ISR in each exception vector. Unlike interrupt vector locations that can be changed, however, exception vectors are permanently assigned by the hardware.

2.3.6 Software Interrupts

Software interrupts are program-invoked instructions like subroutine calls, but are handled by the 8088 processor just like hardware interrupts. An *interrupt* instruction has the form

INT *n*

where *n* is an interrupt type. When this instruction is executed, the processor behaves as if a hardware device caused an interrupt with interrupt type *n*, resulting in execution of the code whose segment : offset address is in interrupt vector location 4*n*. A software interrupt is executed regardless of the value of the interrupt flag in the FLAGS register. When the ISR returns with an IRET instruction, control returns to the instruction following the INT instruction.

Software interrupt mechanisms are often employed by operating systems to permit user access to system services without the user needing to know the address of the service subroutine. It allows the ISR code to be modified or relocated without having to make changes in user code. The chief disadvantage of software interrupts is that the interrupt instruction (INT) and the corresponding return from interrupt (IRET) instruction are more time-consuming to execute than simple subroutine call and return instructions.

Low-level system services provided by the ROM BIOS are accessed through software interrupts, as we shall see in the next section.

2.3.7 The ROM BIOS

The PC ROM BIOS (Basic Input/Output System) plays several roles. For our purposes, it provides a *virtual machine* which insulates us somewhat from the actual peculiarities of the hardware. PC-Xinu will run on any PC system that supports the standard BIOS calls and hardware interrupts described below. This is not without a penalty, however. BIOS calls are not particularly efficient; for example, the BIOS keyboard ISR buffers keyboard characters prior to PC-Xinu doing the same. BIOS calls are also not *re-entrant*, meaning that multiple processes cannot be executing such calls simultaneously.

The BIOS contains code which is executed when the the power to the PC is turned on. This code sets up the standard device interrupt vectors, does a system integrity test and memory check, and then reads and starts execution of the operating system stored on disk. The BIOS contains the ISRs for the keyboard, disk, and clock which are necessary for system operation.

The BIOS also provides user-accessible routines for accessing keyboard characters (KBD), writing to the video display (VID), and carrying out disk transfer operations (DSK). These routines are accessed through software interrupts and are summarized at the end of this section. Note that the BIOS provides a number of additional services which will not be discussed here.

Most BIOS software interrupts have subfunctions which provide specific BIOS services for the particular device. For example, one subfunction for KBD returns information about whether there are any characters in the keyboard buffer, and a second subfunction returns the next character from the keyboard buffer. The subfunction number is conventionally loaded into the AH register prior to making the INT call. Other information is passed to the BIOS routine through registers. Information returned to the user after making a BIOS call is stored in specific registers or in bits in the FLAGS word. Aside from the use of register AH for passing a subfunction code, there is no standard meaning for register use among the different BIOS routines. PC technical documentation states that BIOS calls preserve all registers not specified in the calling sequence or return registers. Specific register conventions for each of the standard BIOS routines discussed in this book are given in subsequent sections.

2.3.8 BIOS Interrupt Service Routines

The BIOS handles standard device and exception interrupts. Upon system startup, the device and exception interrupt vectors are loaded with the segment : offset addresses of BIOS ISRs which are used to service these devices.

It is possible for a user program to *revector* a device interrupt to another, non-BIOS ISR. To do this, it is only necessary to load the segment : offset address of the new ISR into the device vector location, after which all the device interrupts will be handled by the new ISR. Loading this address should be carried out with processor interrupts disabled to avoid the problem of an interrupt occurring in the midst of changing the address.

BIOS services will not function properly if device interrupts are revectored to user programs. For example, if keyboard interrupts are revectored and serviced only by a user ISR, the BIOS KBD routines will never have the opportunity to determine when keystrokes have occurred and, consequently, will not be able to put characters in the BIOS keyboard buffer. On the other hand, if keyboard device interrupts are not revectored, a user program may not know when an interrupt has occurred and may be forced to poll the keyboard device to determine if a character is available. The result is a needless and time-consuming execution of code.

The solution to the revectoring dilemma is to observe that the user program needs to be *informed* when a device interrupt has occurred and to allow the BIOS ISR to handle the interrupt as it usually does. To do this, the device interrupts should be revectored to a user ISR that marks the presence of the interrupt but then immediately *calls* the standard BIOS ISR to service it. When the BIOS ISR returns to the user ISR, the user code will know that an interrupt has occurred, can call BIOS functions to extract buffered data, or can initiate other system activities which depend upon the occurrence of the interrupt.

Two device interrupts that will be revectored in this way are the keyboard (KBD_INT) and clock (CLK_INT). These devices will be discussed in Section 2.4.

2.3.9 Pseudo-device interrupts

In the discussion of software interrupts, we observed that an INT instruction can cause an interrupt action which behaves exactly like a hardware interrupt. When this happens through the BIOS as the result of an asynchronous event such as a 'real' device interrupt, we call it a *pseudo-device* interrupt. The BIOS initiates pseudo-device interrupts upon receiving certain keystrokes. For example, receipt of the keyboard combination *Ctrl-Break* results in a type 1BH (hexadecimal†) software interrupt issued by the BIOS. PC-Xinu revectors this interrupt and terminates execution upon its receipt after restoring all the revectored interrupts back to their previous states. Note that this interrupt should *not* call the standard ISR first, since chaos may result if PC-Xinu loses control forever without having the chance to restore its revectored interrupts.

2.3.10 Interrupt Vector Summary

Figure 2.6 summarizes the device interrupts that PC-Xinu monitors and the BIOS services it uses. More information about specific device ISRs will be discussed in the sections below.

Interrupt Type	Name	Description
08H	CLK_INT	clock hardware interrupt
09H	KBD_INT	keyboard hardware interrupt
10H	VID	video display BIOS service
13H	DSK	disk BIOS service
16H	KBD	keyboard BIOS service
1BH	KBD_BRK	Ctrl-Break pseudo device

Figure 2.6 BIOS interrupts

2.4 Standard PC I/O Devices

A typical PC system has hardware devices for keyboard input, video output, mass storage, and a real-time clock. Additional devices on some systems may include parallel or serial ports for modems, printers, or pointing devices such as a mouse, or for high-resolution graphics displays. We will assume a standard configuration consisting of a *keyboard*, a *video display*, a single *floppy disk*, and a *real-time clock*.

† Throughout the text we will use the assembly language convention that constants with a trailing 'H' are hexadecimal. In the C programming environment, we will adhere to the C convention that constants with a leading 0x are hexadecimal and constants with a leading 0 are octal. As noted before, segment : offset addresses will always be assumed hexadecimal.

2.4.1 Keyboard

A keyboard interrupt occurs whenever a key is pressed or released. This interrupt (KBD_INT) has type 09H and, consequently, has interrupt vector at location 0000:0024. The scan code of the key that was pressed or released is read from the keyboard data *port*.

When an alphabetic key is pressed, its scan code is used to look up its corresponding ASCII character code which is then deposited in RAM into a small BIOS keyboard buffer. Some keys, such as 'Shift' or 'Ctrl', have different effects when they are pressed than when they are released, and others, such as 'Caps Lock', result in a ''toggle'' action. Information about the states of these keys is kept in a fixed RAM location known to the ROM BIOS.

PC-Xinu revectors keyboard interrupts of type 09H to its own code space. But the task of handling interrupts for both pressing and releasing keys, as well as translating scan codes into ASCII codes, is already done by the ROM BIOS and, except for the sake of efficiency, it is unnecessary for PC-Xinu to duplicate this effort. Consequently, upon receipt of the interrupt, PC-Xinu first calls the BIOS keyboard ISR to retrieve the scan code from the keyboard port and to deposit a character in the BIOS keyboard buffer if appropriate. Upon return from the BIOS call, the PC-Xinu keyboard ISR awakens a special-purpose process whose job is to retrieve the character from the BIOS keyboard buffer.

Characters are deposited by the BIOS in the keyboard buffer as 16-bit *key codes*. Typically, the key code of a key is composed of the scan code (obtained from the keyboard) in the high byte and the ASCII code of the key in the low byte. If the low byte is zero, the key code corresponds to one of several *special* keys such as function keys or keypad keys which do not have ASCII counterparts.

Two KBD subfunctions are used to retrieve characters from the keyboard buffer. Subfunction KBDPEND conditionally sets the Z-flag (ZF) in the FLAGS register depending on whether there is a character in the keyboard buffer. If ZF=1 upon return, the keyboard buffer is empty. If ZF=0 upon return, the keyboard buffer is nonempty and AX contains the first character in the buffer.

Subfunction KBDGETC of KBD removes the first character in the keyboard buffer and returns with its key code in register AX. If the keyboard buffer is empty, the routine waits (using a busy-wait loop) until a character is ready. Notice that PC-Xinu avoids this busy-wait by making this subfunction request only if it has detected the presence of a character in the keyboard buffer using subfunction KBDPEND.

Figure 2.7 gives a summary of these KBD subfunctions.

INT KBD (KBD=16H)

Subfunction AH	Name	Description	Registers
00H	KBDGETC	Read next character in keyboard buffer (busy wait if buffer is empty)	returns: AX=key code
01H	KBDPEND	Test for character in keyboard buffer	returns: ZF=1 if no key in buffer ZF=0 & AX=key code

Figure 2.7 Keyboard I/O subfunctions

2.4.2 Video Display

Characters are displayed on the video display by special-purpose hardware which reads ASCII codes and character attributes from specific RAM locations and displays the information in human-readable form on a graphics display device such as a CRT. Describing the actual hardware devices used in the video display is beyond the scope of this book.

Each character position on the video display corresponds to a 16-bit video RAM location; the low byte contains the ASCII code for the character (or special graphics symbol) to be displayed, and the high byte contains attribute information. The attribute byte determines how the character or symbol is to be displayed and how the background will appear; attributes include color, intensity, and blink. Of course, monochrome displays do not support color attributes.

The layout of the attribute byte is given in Figure 2.8.

Bits:	7	6-5-4	3	2-1-0
Contents:	blink	background	intensity	foreground

Figure 2.8 Bits of a video attribute byte

For normal white-on-black display the three foreground bits will represent 7 (binary 111), and the remaining bits will be 0.

A 25-line, 80-column display has 2000 (decimal) character locations and, therefore, requires 4000 bytes of video memory to display one complete screenful of text. Observe that even blanks are considered as ‘characters’, so an all-blank screen still requires 4000 bytes of memory. The character/attribute bytes are stored in consecutive video RAM memory locations on a row-by-row basis, and from left-to-right in each row. Thus the leftmost character at the top row of the screen appears as the first character/attribute pair in video RAM, and the rightmost character at the bottom of the screen appears as the last pair in video RAM.

It is possible to change the contents of the video display by writing directly to video RAM memory locations. This technique is often the fastest way to display text information on the screen. However, the ROM BIOS provides video display (VID) services which accomplish the same task and, in addition, provides other services such as positioning the cursor on the screen and erasing and scrolling portions of the screen. Figure 2.9 summarizes the VID subfunctions used by PC-Xinu.

INT VID (VID=10H)

Subfunction AH	Name	Description	Registers
00H	SETMOD	Set video mode	AL=2 for monochrome AL=3 for color
02H	SETCSR	Set cursor position	DH=row (0 .. 24) DL=col (0 .. 79) BH=[0]
05H	SELADP	Select display page	AL=[0]
06H	SCRLUP	Scroll/clear window	AL=no. of lines to scroll (AL=0 to clear entire window) CH,CL=top left row,column DH,DL=bottom right row,col BH=attr. of blanked lines
0AH	WCHR	Write char at current cursor position	AL=character BH=[0] CX=[1]
0EH	WTTY	TTY write (handles scroll, special chars)	AL=character BH=[0] BL=[7]

Figure 2.9 Video IO subfunctions

The values shown in brackets are the ones PC-Xinu uses; the other values depend on the request. Conventional screen output is possible with subfunction 0EH (WTTY). The other subfunctions are used in the window environment described in Chapter 14.

2.4.3 Floppy Disk

Computer systems often use magnetic storage devices for storage of bulk information. The name *disk* implies that the recording surfaces are shaped in round, flat platters that spin. The platter surfaces are coated with a material that records magnetic fields just as a magnetic recording tape does. The mechanism that changes or senses this magnetic coating is called a *read-write head*. A mechanical *arm*, actuated electrically, positions the head(s) to a specified location on the disk, and analog sensing hardware reads or writes data as the disk spins under the head. To increase capacity, multiple platters are often

built on one spinning rod, a *spindle*, accessed by an arm with multiple heads attached. Normally, disk heads do not contact the magnetic surface; they ''fly'' incredibly close to it on a cushion of air. Accidents that occur when mechanical shock or dust particles on the surface cause the heads to bounce up and down and ruin the magnetic surface are called *head crashes*.

Figure 2.10 A disk arm with two read/write heads.

At a given arm position, the surface area under one head forms a ring called a *track*. The set of all tracks for a given arm position is called a *cylinder* because the tracks outline a cylinder in 3-dimensional space.

Each track is divided into fixed-length *sectors*; a sector is the smallest unit of storage that can be read or written in a single operation. PC-Xinu software assumes a sector length of 512 bytes. Data transferred to or from a sector is often called a *block*.

Compared to main memory, which operates at speeds measured in nanoseconds, disk devices are slow and awkward. If the drive motor is not running, starting the motor and waiting for it to come up to speed may take 500 milliseconds. Once the motor is running, moving the arm to the correct track requires tens of milliseconds. Even reading a track once the arm is in place is nontrivial because the disk must revolve at least once, which may take another 100 milliseconds. Thus, a disk access can be on the order of a million times slower than a memory access. To put this in perspective, imagine that you need to retrieve a pencil before you continue some task. Suppose it takes 10 seconds to walk to your desk (high speed memory) and retrieve one. A retrieval operation that took a million times as long would last roughly 115.7 days. After a few such retrieval operations, you would quickly learn to save pencils in your desk drawer to avoid the trip.

Operating systems use an analogous technique – they often contain complex algorithms that save copies of frequently-used blocks to avoid unnecessary disk accesses.

If disks are so slow, why use them at all? Disk storage offers two important features that main memory does not. First, disk storage is less expensive byte-per-byte. Second, it provides permanent, long-term storage. Unlike main memory which is called *volatile* because data ''evaporates'' when power is removed, data on disk is *nonvolatile*; it persists even if power to the system is turned off.

Disk hardware is much more complex than the keyboard and video devices described above. In addition to the *disk drive*, which consists of the physical arm and read/write head(s), disk hardware includes a complicated electronic *controller*.

A disk controller, which often contains a microprocessor itself, receives commands from the CPU to position the disk arm, to control transfer of data, and to return status information if any errors occur. A controller can operate multiple disk drives (although it may not be able to transfer data to more than one disk simultaneously).

The most significant difference between the disk and keyboard hardware is this:

> *Unlike keyboard hardware which interrupts the CPU for each key pressed or released, the disk interface transfers large blocks of data directly to or from memory without direct intervention by the CPU.*

The technique of transferring large blocks of data without using the CPU is called *direct memory access* (DMA). In the PC, DMA is carried out by a special-purpose 8237 DMA controller chip. This chip disables the CPU while a DMA transfer is taking place and permits the CPU to continue when the transfer is complete. In more sophisticated systems, DMA transfers can take place at the same time as the CPU is executing instructions. In such systems, system throughput is increased dramatically by allowing simultaneous data transfer to several peripherals while the CPU continues to execute instructions.

2.4.4 The Pieces Of Disk Hardware

The disk hardware has no notion of *file* or *directory* built in; these are added by the system software, as we will see in Chapter 17. For a specific drive, the hardware considers each physical sector of disk data to be addressed by a *physical sector address*, which is a tuple of the form

(cylinder, surface, sector)

The *cylinder* is the number of the cylinder on the drive, starting with zero as the outermost cylinder. *Surface* specifies which read/write head is to be used. A double-sided disk has two surfaces numbered zero and one; a multi-platter disk has more than two surfaces. Note that the combination (cylinder, surface) specifies a track. Finally, the *sector* designates which sector of the specified track is to be used; sector numbers within a track usually start with one.

In the standard PC configuration, a floppy disk consists of 40 cylinders (numbered 0 to 39), 2 surfaces (numbered 0 and 1) per cylinder, and 9 sectors (numbered 1 to 9) per track. This gives a total of 720 sectors per drive. Since each sector contains 512 bytes, the total capacity of a disk is 368,640 bytes, or 360K.

2.4.5 Logical sector numbers

It is convenient to think of each of the 720 sectors of a disk as having a unique sector number in the range 0 to 719, which we will call a *logical sector number*. However, the disk controller interface requires physical sector addresses in (cylinder, surface, sector) format. Not just any correspondence between logical sector numbers and sector addresses will do, since moving the disk arm is a particularly time-consuming operation, and disk requests frequently cluster together in logically adjacent sectors. The guiding principle is thus:

> *Logical sector numbers should map to physical sector addresses in such a way that consecutive logical sector numbers map as often as possible into physical sector addresses that lie on the same track.*

The simplest way to carry out this principle is to map the first logical sector numbers 0 through 8 into physical sector addresses (0,0,1) through (0,0,9), 9 through 17 into (0,1,1) through (0,1,9), 18 through 26 into (1,0,1) through (1,0,9), etc. In general, given a logical sector number, *lsn*, the mapping into (*cylinder, surface, sector*) is given by

```
sector = (lsn % 9) + 1;
lsn = lsn / 9;
surface = (lsn % 2);
lsn = lsn / 2;
cylinder = lsn;
```

where '%' means the modulus operator.

The hardware reads data by copying a sector on the disk to a block in memory, and writes data by copying a block from memory onto a sector of the disk. Using the disk hardware consists of passing read or write requests to the BIOS disk function DSK, which carries them out and reports the results. As with the keyboard and video display devices, the read and write requests are subfunctions of the DSK function. These subfunctions are detailed in Figure 2.11 below.

INT DSK (DSK=13H)

Subfunction AH	Name	Description	Registers
02H	DSKREAD	Read sector	AL=[1] CL=sector no. (1-9) CH=cylinder (0-39 decimal) DL=drive (0-no. of drives) DH=surface (0-1) ES:BX=transfer address see below for return values
03H	DSKWRITE	Write sector	same as above

returns:
CF (carry flag) = 0 for successful operation
CF = 1 for failure & AH=error code (hex):
02H=invalid sector ID
03H=write protect error
04H=record not found
08H=DMA overrun
09H=access across 64K boundary
10H=CRC error
20H=controller failure
40H=seek failure
80H=timeout - disk failed to respond

Figure 2.11 Diskette I/O subfunctions

The register value for AL in brackets is used by PC-Xinu software; the values in the other registers depend on the request.

2.4.6 Real-Time Clock

A real-time hardware clock is one of the most important devices to an operating system. A clock enables the operating system to ensure that processes take no more than their share of processor time. A clock also allows the operating system to provide time-related services to processes, including keeping track of the current time-of-day. We will discuss the role of a real-time clock in PC-Xinu in more detail in Chapter 10.

The PC contains a clock circuit having a frequency of 1.19318MHz. The output from this circuit is connected to an 8253 timer chip which divides this frequency by 65536 to provide an interrupt approximately 18.2 times per second. This interrupt has type 08H and has its vector at location 0000:0020.

At each clock interrupt, the BIOS interrupt service routine for the clock increments a 32-bit time-of-day counter. In addition, the BIOS ISR monitors the state of the diskette motor and turns off the motor if it has been on for about two seconds without an intervening diskette I/O request.

Similar to keyboard interrupts, PC-Xinu revectors clock interrupts of type 08H to its own code space. Upon receipt of the interrupt, PC-Xinu first calls the BIOS clock ISR. When the BIOS call returns, PC-Xinu carries out its clock-specific activities.

2.5 The C Run-Time Environment

Operating systems should be written in high-level languages because it makes them easier to write, understand, debug, and move to other machines. We have chosen the C programming language for PC-Xinu. C is a concise and powerful systems language that is well-suited to operating system implementation. It allows the programmer to manipulate addresses and specify storage layouts. It is also a high-level language that supports parameterized procedures, reasonable control statements, and separate compilation.

For the most part, it is possible to write an operating system without knowing the details of the compiler, the code it produces, or the conventions used by that code when it runs. From time to time, however, it will be necessary to manipulate the underlying run-time environment. This section reviews a few of the pertinent details.

The C compiler expects that each program will be run in an address space laid out in two *segments*. Figure 2.12 shows the arrangement of segments at run-time.

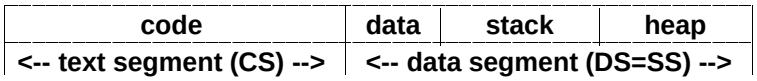

Figure 2.12 Storage layout for a C Program

The *text segment*, which includes code for the main program and all procedures, occupies the lower part of the address space and is referenced using the CS segment register. The *data segment*, which contains program data followed by the stack and heap, occupies the higher region of the address space. Data segment references are made using the DS segment register. The SS (stack) segment register always points to the same segment as the DS register. The stack occupies a fixed-size component of the data segment address space immediately beyond the program data. The stack pointer (SP) initially points to the highest address of the stack and grows downward. The area between the stack and the top of the data segment address space is free space which may be used for run-time storage allocation (the *heap*).

When PC-Xinu runs multiple processes, it allocates a stack area for each of them (but places the text and data for all processes together). These stacks are generally allocated from the bottom of free space. Thus, if three processes start, their stacks are allocated contiguously upward as shown in Figure 2.13:

Figure 2.13 Storage layout when PC-Xinu runs.

Because the hardware does not protect one user from another, stack overflow in one process will, unfortunately, destroy the data in a stack that belongs to another process. (An exercise at the end of Chapter 4 outlines one way to check for stack overflow.) Stack space is returned to the free list whenever a process exits. PC-Xinu also provides a system call that allocates free space to user processes for general use.

There is an important distinction between names in object programs and names in C source programs. It is this:

> *The compiler adds an underscore to all external symbols in C programs.*

Thus, names like *_end* are declared in C without the underscore (i.e., ‘‘extern end’’). The reader must remember this distinction when comparing C to assembler programs in which external symbols have explicit underscore prefixes.

The C procedure calling conventions are extraordinary. During a procedure call, the calling procedure is easiest to understand. It pushes actual arguments on the stack *in reverse order* and then calls the specified routine. Pushing parameters in reverse order allows procedures like *printf* to be called with a variable number of arguments. Even though the called routine cannot determine the number of arguments from the information on the stack, it can always find the first argument by looking on the stack just beyond the return address. Under the C conventions, the calling procedure is also responsible for popping arguments off the stack after the called procedure returns.

The C calling sequence becomes more complex after the called procedure begins. The called procedure is responsible for preserving certain registers that it will use. In particular, the BP, SI and DI registers must be saved upon procedure entry and restored upon exit. Saving SI and DI is necessary only if these registers are used by the procedure.

The BP register performs an important referencing function during procedure execution. Upon procedure entry, the current value of BP is pushed onto the stack and the value of SP is moved into BP. Space is then allocated on the stack to accommodate automatic local variables. Finally, registers SI and DI are pushed on the stack if necessary. (The location where these registers are pushed on the stack may vary from compiler to compiler.)

BP serves as a pointer to the *stack frame* containing the parameters passed by the caller and the procedure’s local variables. At this point the stack frame appears as follows:

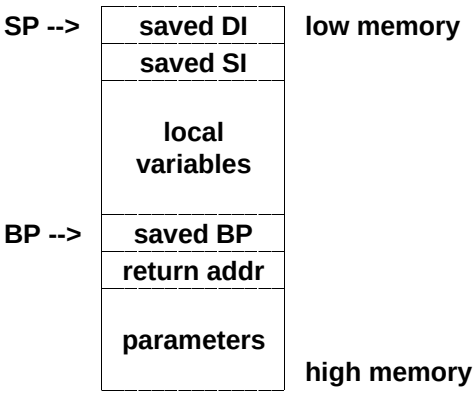

Figure 2.14 C stack frame

During subroutine execution, parameter and local variable values can be determined using fixed offsets to BP.

Return values are deposited into registers before returning to the calling procedure. Integer (16-bit) return values are deposited in the AX register; return values longer than 16 bits use additional general-purpose registers. The called procedure returns to the calling procedure after popping the saved DI and SI registers, deallocating space for local variables, and popping the saved BP register. Note that the parameters remain on the stack upon return to the calling procedure; it is the calling procedure's responsibility to deallocate the parameters on the stack.

2.6 Assembly Language Interface

Assembly language procedures which will be called from C programs must follow the layout of the stack frame during subroutine execution as described in the previous section. The following examples illustrate assembly language procedures and related C procedures which are used in PC-Xinu.

To use these procedures, the assembler must be used to create object modules from the assembly language files, and the C compiler must be used to create object modules from the C program files. The final step to produce executable code is to link the assembled and compiled object modules together with the object module of a main driver program. The main driver program for PC-Xinu will be introduced in Chapter 13. In the meantime, special purpose main programs may be written to serve as test drivers.

2.6.1 Low-Level Keyboard Input

The procedure *kbdgetc* returns the next character from the BIOS keyboard buffer or, if there is no character available, returns NOCH. The constant NOCH is defined in the header file *kbdio.h* and is designed to have a value which cannot be that of an ordinary character.

```
/* kbdio.h */

#define NOCH    -1
#define SPEC    0x100

extern int kbdgetc();                   /* defined in kbdio.asm */
extern int kbdint();                    /* defined in kbdio.asm */
extern int kgetc();                     /* defined in kgetc.c   */
```

File *kbdio.h* introduces features of C and conventions used throughout the book. Because the name ends in *.h*, it implies that this file will be included in other programs (the *h* stands for *header*). Such files often contain the declarations for global data structures, symbolic constants, and external procedures. The *kbdio.h* file is no exception. It declares the functions *kbdgetc*, *kbdint* and *kgetc* to be global (extern), so programs in PC-Xinu will be able to access it. It also defines symbolic constants like NOCH. Constants are referenced by name throughout the code because it helps make their purpose clear.

Kbdgetc checks whether there is a character in the buffer using the KBDPEND subfunction of the KBD BIOS interrupt described in Section 2.3.8 This information is returned in the ZF bit of the FLAGS register. The ZF flag can be tested using the conditional jump instruction JNZ, which jumps to the designated label if ZF is false (0). If no character is present, NOCH is deposited in the AX register. Otherwise the key code is extracted from the BIOS keyboard buffer using the KBDGETC subfunction which returns its value in register AX. As observed in section 2.4.2, a key code with a zero low byte (in AL) corresponds to a special non-ASCII key; in this case, the value returned is the value of the scan code (in AH) plus 100H, the value of SPEC.

If there is a character in the BIOS buffer, *kbdgetc* makes another BIOS call to retrieve the character. Should a second process be permitted to run between the first and second BIOS calls, the second process could retrieve the character from the buffer before the first process had a chance to retrieve it. In this case, when the first process resumes, it will find the buffer empty and will be delayed by the BIOS call. To prevent this from happening, *kbdgetc* clears the *pcxflag* variable before testing for a character in the BIOS buffer. This effectively makes it impossible for any other process to run while *kbdgetc* is being called. The value of *pcxflag* is restored to its original value before *kbdgetc* returns. The role of *pcxflag* is discussed further in Chapters 4 and 9.

Kbdgetc also preserves the FLAGS register using the *pushf* and *popf* instructions. This is necessary because *kbdgetc* may be called when interrupts are disabled, and the KBD interrupt alters the FLAGS register.

In addition to the special-purpose *kbdgetc* function, *kbdio.asm* includes the definition of a general *kbdint* procedure which permits access to all the *KBD* interrupt subfunctions. Further information about these subfunctions may be obtained from the manufacturer's BIOS documentation.

```
; kbdio.asm - _kbdgetc, _kbdint

KBD     equ     16H             ; keyboard request interrupt
KBDGETC equ     0               ; get the character
KBDPEND equ     1               ; check for character pending
NOCH    equ     -1              ; no character
SPEC    equ     100H            ; special character offset

        include dos.asm

        dseg
; dummy data segment
        endds

        pseg

        public  _kbdgetc
        public  _kbdint
        extrn   pcxflag:word

;---------------------------------------------------------------------------
; _kbdgetc  --  get a character from the BIOS keyboard buffer
;---------------------------------------------------------------------------
; int kbdgetc()
_kbdgetc        proc    near
        pushf                           ; push the flags
        push    si
        push    di                      ; save registers
        cli                             ; disable interrupts
        xor     ax,ax                   ; to defer rescheduling, ...
        xchg    ax,cs:pcxflag           ; ... get and clear pcxflag ...
        push    ax                      ; ... and save for later
        mov     ah,KBDPEND              ; get keyboard status first
        int     KBD
        jnz     getc1                   ; character there?
        mov     ax,NOCH                 ; if not, send the info back
```

```
        jmp     short getc9
getc1:  mov     ah,KBDGETC              ; if so, actually get the char
        int     KBD
        or      al,al                   ; check the lower byte
        je      getc2                   ; is it a non-ASCII special?
        xor     ah,ah                   ; if not, just send the lower byte
        jmp     short getc9
getc2:  mov     al,ah                   ; move scan code to lower byte
        xor     ah,ah                   ; clear out upper byte
        add     ax,SPEC                 ; add special offset
getc9:  pop     cs:pcxflag              ; restore pcxflag
        pop     di
        pop     si                      ; restore registers
        popf
        ret
_kbdgetc        endp

;--------------------------------------------------------------------------
; _kbdint  --  general access to KBD interrupt; returns flags
;--------------------------------------------------------------------------
; int kbdint(r)
; union REGS *r;
_kbdint proc    near
        push    bp
        mov     bp,sp                   ; follows C calling conventions
        pushf                           ; save flags
        push    si
        push    di                      ; save registers
        mov     si,[bp+4]               ; get ptr. to register structure
        mov     ax,[si]
        mov     bx,[si+2]
        mov     cx,[si+4]
        mov     dx,[si+6]               ; set up registers for call
        push    si
        int     KBD                     ; call the KBD BIOS interrupt
        pop     si                      ; recover register pointer
        pushf                           ; save flags for return value
        mov     [si],ax
        mov     [si+2],bx
        mov     [si+4],cx
        mov     [si+6],dx               ; return registers
        pop     ax                      ; return flags value
        pop     di
        pop     si                      ; restore registers
```

```
        popf                            ; and the flags
        pop     bp
        ret
_kbdint endp

        endps

        end
```

Kgetc is C procedure which is companion to *kbdgetc*. It calls the assembly language procedure *kbdgetc* to retrieve a value from the BIOS keyboard buffer, looping for another value as long as *kbdgetc* returns NOCH. If the character returned by *kbdgetc* is a carriage return ('\r'), *kgetc* translates it into a newline ('\n'); otherwise *kgetc* returns the character unchanged to the caller.

```
/* kgetc.c - kgetc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <kbdio.h>

/*------------------------------------------------------------------------
 *  kgetc  --  get the next character from the keyboard
 *------------------------------------------------------------------------
 */
int kgetc(d)
int d;                                  /* dummy parameter              */
{
        int     ch;

        while ( (ch=kbdgetc()) == NOCH )
                ;
        return ( (ch==RETURN) ? NEWLINE : ch );
}
```

The loop involving NOCH at the beginning of this procedure violates the prohibition of ‘‘busy waiting’’ given in Section 1.4.7 of Chapter 1. While, in general, busy waiting is not to be tolerated for low-level procedures in an operating system, *kgetc* is intended for use only for system debugging or for low-level system interaction when interrupts are disabled. More general keyboard routines will be discussed in Chapter 12. Notice that *kgetc* has a dummy *int* parameter used to make its calling sequence consistent with the generic *getc* procedure defined in Chapter 11.

2.6.2 Low-Level Video Display

The *wtty* procedure displays a character at the current cursor position on the screen. It uses the WTTY subfunction of the VID interrupt described in section 2.4.2 The file *vidio.h* contains the procedure declarations.

```
/* vidio.h */

extern void wtty();                     /* defined in vidio.asm         */
extern int vidint();                    /* defined in vidio.asm         */
extern void kputc();                    /* defined in kputc.c           */
```

The assembly language code is straightforward. Observe that *wtty* preserves the SI and DI registers. While the standard BIOS documentation states that registers (except those used specifically for parameter passing) are preserved by BIOS interrupts, the particular implementation used for developing these programs failed to preserve them – a fact that resulted in several frustrating hours of debugging.

Vidio.asm also contains the definition of the *vidint* procedure, allowing access to all the *VID* subfunctions. This procedure will be used in Chapter 14 for performing window functions.

```
; vidio.asm - _wtty, _vidint

        include dos.asm

VID     equ     10h                     ; video interrupt
WTTY    equ     0EH                             ; TTY write subfunction
FORE    equ     7                       ; foreground color
ADP     equ     0                       ; active display page

        dseg
; null data segment
        endds

        pseg

        public  _wtty
        public  _vidint

;---------------------------------------------------------------------------
; _wtty  --  put a character to the video display using WTTY call
;---------------------------------------------------------------------------
; void wtty(ch)
```

```
; char ch;
_wtty   proc    near
        push    bp                      ; follows C calling conventions
        mov     bp,sp
        pushf                           ; push the flags
        push    si                      ; push registers since . . .
        push    di                      ; . . . INT VID clobbers these
        mov     al,[bp+4]               ; character to write
        mov     ah,WTTY                 ; TTY write subcommand
        mov     bl,FORE                 ; foreground color
        mov     bh,ADP                  ; display page
        int     VID                     ; call the video function
        pop     di
        pop     si                      ; restore registers
        popf
        pop     bp
        ret
_wtty   endp

;--------------------------------------------------------------------------
; vidint  --  general access to VID interrupts; returns flags
;--------------------------------------------------------------------------
; int vidint(r)
; union REGS *r;
_vidint proc    near
        push    bp
        mov     bp,sp                   ; follows C calling conventions
        pushf                           ; save flags
        push    si
        push    di                      ; save registers
        mov     si,[bp+4]               ; get pointer to reg. structure
        mov     ax,[si]
        mov     bx,[si+2]
        mov     cx,[si+4]
        mov     dx,[si+6]               ; set up registers for call
        push    si
        int     VID                     ; call the VID BIOS interrupt
        pop     si                      ; recover register pointer
        pushf                           ; save flags for return value
        mov     [si],ax
        mov     [si+2],bx
        mov     [si+4],cx
        mov     [si+6],dx               ; return registers
        pop     ax                      ; return flags value
```

```
        pop     di
        pop     si                      ; restore registers
        popf                            ; and the flags
        pop     bp
        ret
_vidint endp

        endps

        end
```

Companion to *wtty* is the C procedure *kputc*. Like *kgetc*, *kputc* calls the assembly language routine *wtty* to do the actual character output and makes a translation of newlines ('\n') into return+newline combinations ('\r' and '\n'). The first parameter to *kputc* is a dummy *int* parameter used to make its calling sequence consistent with the generic *putc* procedure described in Chapter 11. The other parameter is the character to display.

```
/* kputc.c - kputc */

#include <conf.h>
#include <kernel.h>
#include <tty.h>
#include <vidio.h>

/*------------------------------------------------------------------------
 *  kputc  --  put a character to the video display
 *------------------------------------------------------------------------
 */
void kputc(d,ch)
int d;                                  /* dummy parameter              */
char ch;                                /* character to display         */
{
        if ( ch==RETURN || ch==NEWLINE ) { /* expand newline            */
                ch = NEWLINE;
                wtty(RETURN);
        }
        wtty(ch);
}
```

2.6.3 Low-Level Disk Services

The disk assembly language procedure *dskio* performs low-level read and write operations to the disk. The operation to perform (DSKREAD or DSKWRITE), the disk number, and the physical sector address (cylinder,surface,sector) are passed as parameters. The constants DSKREAD and DSKWRITE as well as the procedure declarations are defined in the header file *dskio.h*.

```
/* dskio.h */

#define DSKREAD         2               /* DSK read subfunction code    */
#define DSKWRITE        3               /* DSK write subfunction code   */
#define RETRY           2               /* no. of disk retry operations */

#ifndef NDSECT
#define NDSECT          720             /* no. of disk sectors          */
#endif

extern int dskio();                     /* defined in dskio.asm         */
extern int dskint();                    /* defined in dskio.asm         */
extern int dread();                     /* defined in dio.c             */
extern int dwrite();                    /* defined in dio.c             */
```

Also passed to *dskio* is the transfer address, which is the memory address used for the disk transfer. The number of bytes to transfer is always one block (512 bytes).

In reading this code, recall that the DSKREAD and DSKWRITE subfunctions of the DSK BIOS service require the transfer address to be ES:BX. Since the transfer address (passed to DSKREAD and loaded into BX) is an offset to the data segment DS, it is necessary to load the ES segment register with the same value as DS prior to calling the DSK interrupt. *Dskio* passes back to the calling procedure the error code returned in AH by the DSK interrupt call. A zero return indicates no error. The *dskio.asm* file also contains the definition of the general *dskint* procedure, providing access to all the *DSK* subfunctions.

```
; dskio.asm - _dskio, _dskint

        include dos.asm

DSK     equ     13H                     ; disk i/o BIOS function

        dseg
; null data segment
        endds

        pseg

        public  _dskio
        public  _dskint

;---------------------------------------------------------------------------
; _dskio  --  perform disk read/write operation
;---------------------------------------------------------------------------
; int dskio(op,buf,drive,cyl,surf,sect)
; int op;                       /* 2=read, 3=write                      */
; char *buf;                    /* transfer address                     */
; int drive;                    /* disk drive number                    */
; int cyl,surf,sect;            /* (cylinder,surface,sector) disk addr  */
_dskio  proc    near
        push    bp                      ; set up the stack frame
        mov     bp,sp                   ; stack frame pointer
        pushf                           ; push the flags
        push    si
        push    di                      ; save registers
        mov     ah,[bp+4]               ; operation code in ah
        mov     bx,[bp+6]               ; buffer pointer in bx
        mov     dl,[bp+8]               ; drive number in dl
        mov     ch,[bp+10]              ; cylinder number in ch
        mov     dh,[bp+12]              ; surface in dh
        mov     cl,[bp+14]              ; sector number in cl
        mov     al,1                    ; transfer one block
        push    ds
        pop     es                      ; set es to our data segment
        int     DSK                     ; call the DSK BIOS interrupt
        mov     al,ah                   ; error return in al
        xor     ah,ah                   ; clear upper byte
        pop     di
        pop     si                      ; restore registers
        popf                            ; restore the flags
```

```
        pop     bp
        ret
_dskio  endp

;--------------------------------------------------------------------------
; _dskint  --  general access to DSK interrupt; returns flags
;--------------------------------------------------------------------------
; int dskint(r)
; union REGS *r;
_dskint proc    near
        push    bp
        mov     bp,sp                   ; follows C calling conventions
        pushf                           ; save flags
        push    si
        push    di                      ; save registers
        mov     si,[bp+4]               ; get pointer to register structure
        mov     ax,[si]
        mov     bx,[si+2]
        mov     cx,[si+4]
        mov     dx,[si+6]               ; set up registers for call
        push    ds
        pop     es                      ; set es to our data segment
        push    si
        int     DSK                     ; call the DSK BIOS interrupt
        pop     si                      ; recover register pointer
        pushf                           ; save flags for return value
        mov     [si],ax
        mov     [si+2],bx
        mov     [si+4],cx
        mov     [si+6],dx               ; return registers
        pop     ax                      ; return flags value
        pop     di
        pop     si                      ; restore registers
        popf                            ; and the flags
        pop     bp
        ret
_dskint endp

        endps

        end
```

Logical sector numbers are passed to the C procedures *dread* and *dwrite* in *dio.c*, which are in turn translated using the mapping given in section 2.4.5 into (cylinder, surface, sector) components and passed on to *dskio* through an auxiliary function *dio*. If *dskio* returns a nonzero error code, *dio* retries the read or write operation up to RETRY times in hopes that the drive controller or physical disk device can recover from a transient error condition. RETRY should not be large, since transient conditions are not generally repeatable and persistent errors are not likely to go away. Both *dread* and *dwrite* return the error code obtained from the *dio* call. If *dio* detects a sector number that is out of range, it returns *SYSERR*, which is defined in the header file *kernel.h*. SYSERR will be used throughout PC-Xinu as a return value indicating an error condition. We will introduce *kernel.h* in Chapter 5.

Observe that *dio* is declared as *LOCAL* in the file *dio.c*. *LOCAL* is expanded into the *static* keyword because of the macro definition in *kernel.h*. The only difference between a *static* procedure and other procedures is that the name of the procedure is not considered as external to the module in which it is defined. This means that the procedures *dread* and *dwrite* are accessible outside of the module *dio.c*, but *dio* is not.

```
/* dio.c - dread, dwrite */

#include <kernel.h>
#include <dskio.h>

LOCAL int dio();

/*------------------------------------------------------------------------
 *  dread  --  read a sector from disk
 *------------------------------------------------------------------------
 */
int dread(buf,dskno,lsn)
char *buf;                              /* transfer address             */
int dskno;                              /* disk number                  */
int lsn;                                /* logical sector no.           */
{
        return(dio(DSKREAD,buf,dskno,lsn));
}

/*------------------------------------------------------------------------
 *  dwrite  --  write a sector to disk
 *------------------------------------------------------------------------
 */
int dwrite(buf,dskno,lsn)
char *buf;                              /* transfer address             */
int dskno;                              /* disk number                  */
int lsn;                                /* logical sector no.           */
```

```
{
        return(dio(DSKWRITE,buf,dskno,lsn));
}

/*------------------------------------------------------------------------
 *  dio  --  translate logical sector numbers to physical disk addresses
 *------------------------------------------------------------------------
 */
LOCAL int dio(op,buf,dskno,lsn)
int     op;                             /* operation code               */
char    *buf;                           /* transfer address             */
int     dskno;                          /* disk number                  */
int     lsn;                            /* logical sector no.           */
{
        int     cyl,surf,sect;
        int     i;
        int     status;

        if ( lsn<0 ||  lsn>=NDSECT )
                return(SYSERR);
        sect = lsn % 9;                 /* sector number                */
        sect = ( 5*sect ) % 9;          /* sector interleaving          */
        lsn /= 9;
        surf = lsn % 2;                 /* surface                      */
        lsn /= 2;
        cyl = lsn;                      /* cylinder                     */
        for (i=0; i<RETRY; i++)
                if ( (status=dskio(op,buf,dskno,cyl,surf,sect+1)) == 0 )
                        break;
        return (status);
}
```

2.7 Summary

We have reviewed the architecture of the PC computer and the 8088 processor. The PC is a conventional microcomputer system, organized around a motherboard and bus through which the CPU accesses memory and devices. Output (input) is performed by writing to (reading from) ports. Interrupt-driven devices cause the CPU temporarily to suspend the current program and to execute an interrupt service routine. The ROM BIOS provides standard device interrupt service routines and low-level services to user programs through software interrupts. Simple devices like the keyboard and the video display require the CPU to transfer one character at a time. More complicated devices like disks can perform I/O a block at a time. They only need the CPU to start a block

transfer. Transmission is carried out by direct memory access (DMA) where the device uses the bus to interact directly with memory.

This chapter also reviewed the C programming language run-time environment and calling conventions. At run-time, the text segment precedes the data segment. The data segment contains program data, followed by the stack and heap. Procedure calls push parameters on the stack in reverse order.

The chapter closed with low-level assembly language routines for keyboard input, video character output, and disk read/write access. These routines illustrate software interrupts as well as the interface between C and assembly language procedures.

FOR FURTHER STUDY

More information on the PC and the 8088 can be found in the vendor's handbooks. Myers [1978], Stone [1972], and Tanenbaum [1976] all provide general discussions of computer architecture; Stone [1975] looks at memory addressing in more detail. These books also review procedure calling conventions, as does Knuth [1968]. The C language calling conventions are described by Johnson and Ritchie [1981]. The Microsoft C calling conventions are detailed in the *Microsoft C Compiler User's Guide*. For Turbo C calling conventions, see the *Turbo C User's Guide*.

EXERCISES

2.1 Compare the 8088 to Digital Equipment Corporation's PDP 11 computers. What are the most important differences?

2.2 Find out about memory management on microprocessors like the 80286 and 80386 or a Motorola 68000-based system. Compare these to the 8088.

2.3 Device interrupt vector addresses are called *programmable* if they can be changed after the hardware has been purchased from the vendor. Why are programmable vector addresses needed?

2.4 Because the disk interface accesses memory directly during DMA transfers, curious errors can result. Find out what happens if you read a 512-byte block of data into a memory location that is within 512 bytes of the highest memory address in the data segment.

2.5 Find out why and how disks are *formatted* into sectors.

2.6 If a disk surface contains a physical flaw, it may make one or more sectors unusable. Some operating systems remap blocks on the disk, placing data for flawed blocks on unflawed sectors. Try to discover how your favorite operating system tolerates flawed disks (if it does).

2.7 Build a stand-alone program to format disks and check for flaws. (The ROM BIOS provides a DSK formatting subfunction.)

2.8 Find out what the keyword *register* means in C.

2.9 Write a C program using only *kgetc* and *kputc* (together with the lower-level *kbdgetc* and *wtty* procedures) which will read a character from the keyboard and echo it to the screen. Do not use any other components of PC-Xinu. Analyze what happens when you enter special characters such as *Tab*, *Back Space*, *Return*, *Ctrl-U*, and PC function key *F1*.

2.10 Write a C program using only the procedures described in this chapter to read a specified sector of the disk and to display the byte contents of the sector in hexadecimal and ASCII format. Do not use any other components of PC-Xinu. Read sectors from a PC-formatted disk and analyze their contents.