Understanding device drivers in Operating System/2

by A. M. Mizell

To meet its design goals for multitasking, Operating System/2™ requires a device driver architecture for interrupt-driven device management. A device driver in OS/2™ is affected by the new architecture both in its structure and in its relationship to the system. An OS/2 device driver contains components, such as the Strategy Routine and Hardware Interrupt Handler, which have well-defined responsibilities. The basic form of these components is a FAR CALL/FAR RETURN model. The operating system calls the device driver components to handle certain types of events, such as an application I/O request or a device interrupt. In responding to these events, an OS/2 device driver must cooperate with the operating system to preserve system responsiveness by helping to manage the multitasking of concurrent activities. Since OS/2 uses both the real mode and the protected mode of the system processor to support DOS and OS/2 applications, respectively, the components of an OS/2 device driver must execute in both modes. In this manner, an OS/2 device driver can be viewed as an installable extension of the Operating System/2 kernel. Comparisons between IBM Personal Computer DOS and Operating System/2 are drawn to illustrate differences between device management and device driver architecture.

A device driver is used to manage devices in small-system operating systems; it is a program that manages the transfer of data to and from a particular device. Operating systems use installable device drivers to support a large number of devices.

Operating System/2™ defines a device-driver architecture within its framework of device management. Device drivers are installed in OS/2 through the configuration file, CONFIG.SYS, which is processed during system initialization. To understand how a device driver relates to the system in Operating System/2, it is first necessary to understand the context of the OS/2 device management.

OS/2 device management

Several design points in OS/2 determine system structure; these include supporting the 80286 protected-mode environment, maintaining the 80286 real mode for DOS applications, and providing multitasking.

The OS/2 support of applications in both protect mode and real mode is performed through mode switching. OS/2, however, minimizes the amount of mode switching that must take place. This means that device I/O may be processed in the mode of the requesting application. OS/2 device management is therefore *bimodal.*

The multitasking capabilities of OS/2 require effective use of the system processor for different activities. A program must be able to execute concurrently with a device that is performing an operation. OS/2 device management must therefore have the ability to asyn-

chronously notify an I/O requestor that an I/O operation is complete. Moreover, one or more programs must be able to use *different* devices at the same time. For example, a printer can be active at the same time as the keyboard. Device management in OS/2 is therefore *interrupt-driven.*

OS/2's predecessor, DOS, represents a different perspective on device management. DOS is a single-task system in which activities are serialized. An application I/O request must complete before another activity can take place. DOS device management cannot asynchronously notify the I/O requestor that an I/O operation has completed. Instead, device management in DOS is based on *polled* I/O, where the state of a device operation is repeatedly checked for completion.

The review of polled I/O versus interrupt-driven I/O characteristics will illustrate why interrupt-driven device management is more suitable for OS/2.

Polled I/O. Polled I/O is accomplished by repeatedly executing a set of instructions that check for completion of an I/O operation. A program issues an I/O request, then cycles through a loop of instructions. In the instruction loop, the program checks an indicator that shows the state of the I/O operation being performed by the device. For example, the indicator may be the value at one of the device's I/O ports or the value of a flag that is set when the device's hardware interrupt invokes an interrupt handler. Figure 1 depicts the polled I/O.

Under DOS, a program must perform I/O according to the polled I/O model. When an application program issues an I/O request to DOS, DOS issues a command to the appropriate device driver, which then issues an I/O command to its device. At this point, the device driver must wait for its device to complete the operation before it can determine what action to take next. If the I/O operation fails, the device driver may try to correct the error and retry the operation. If the I/O operation succeeds, the device driver may reuse resources allocated for the operation, such as memory buffers.

The DOS device management is a simple scheme. The program that controls the device also controls the system processor, because the program must continue to execute while waiting for the I/O request to complete. This procedure is acceptable in a single-tasking environment, because the system processor need not be shared among concurrent activities.

Figure 1 Polled I/O

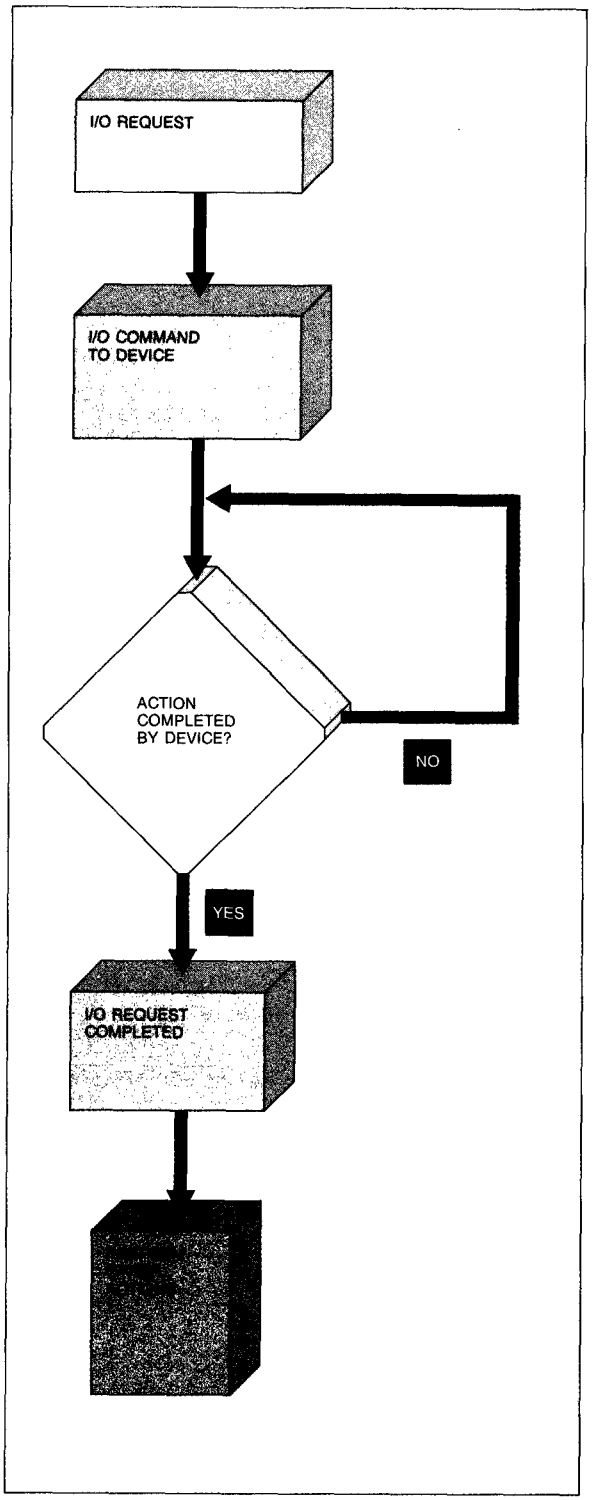
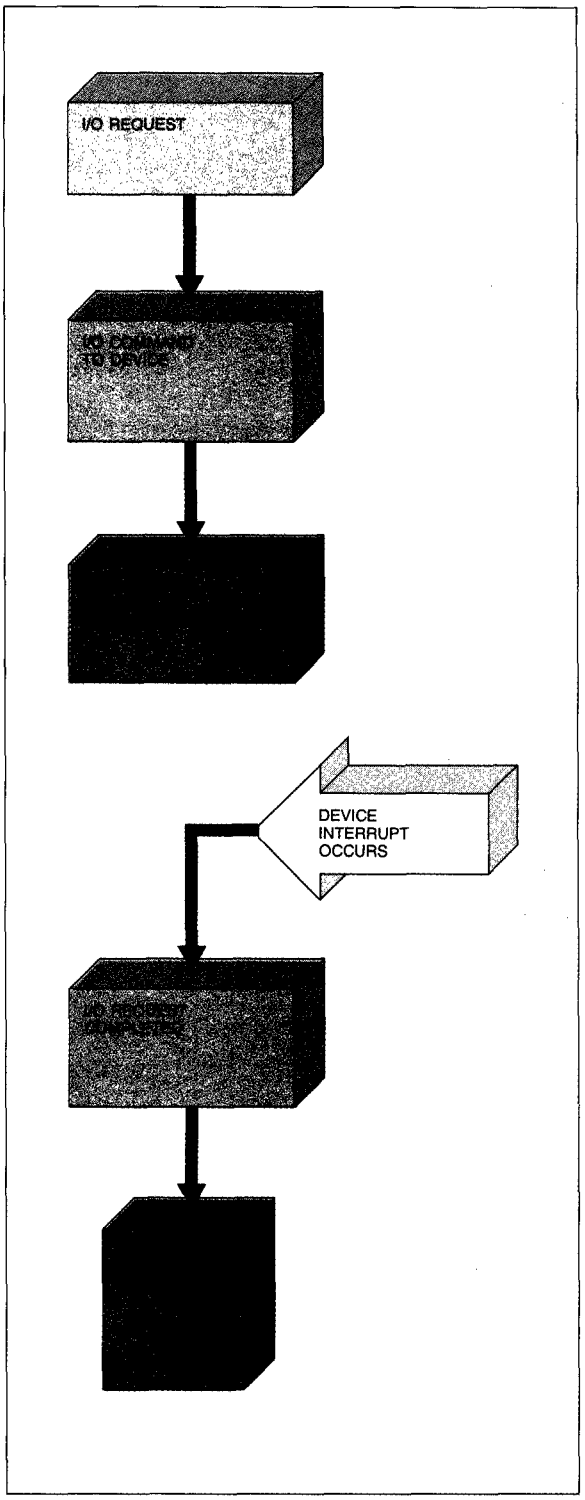


Figure 2 Interrupt-driven I/O



Interrupt-driven I/O. Interrupt-driven device I/O removes the serialization restriction. Interrupt-driven device management supplies a mechanism to asynchronously notify the I/O requestor when the I/O operation has completed. The system processor can be shared among concurrent activities while the I/O operation is being performed by the device.

The heart of interrupt-driven device management is the relationship between OS/2 and its device drivers. The operating system instructs the device driver to perform an I/O operation on behalf of some application thread of execution. The device driver sends the appropriate command to the device and surrenders whatever is left of its time slice for execution. The operating system then dispatches some other thread that is ready to execute. When the device completes the I/O operation, it issues a hardware interrupt to the system. The hardware interrupt asynchronously breaks into the execution of the current thread and causes the device driver's interrupt handler to execute. The device driver's interrupt handler resets the interrupting condition at the device and informs the operating system that the device driver thread waiting on the I/O operation can be dispatched. The device driver thread is then dispatched according to its priority. When it executes, it returns the completion code of the I/O to the operating system to be passed back to the application. Figure 2 depicts the logic for interrupt-driven I/O.

The close interaction between the operating system and its device drivers has several advantages. Different devices can be used concurrently by different applications. Different devices can be used concurrently by a single application (using multiple processes or threads). System performance benefits from intelligent devices (those with specialized microprocessors that offload work from the system processor).

Of course, not all devices produce hardware interrupts; however, those which do can be utilized efficiently by the OS/2 device management. Devices which do not generate interrupts must be inherently fast or managed by the device driver to utilize the processor for insignificant lengths of time.

The role of the device driver

In general, a device driver manages the flow of data to and from its device within the context of application I/O requests.

OS/2 invokes the device driver when an application issues a system request for I/O. In the case where its

device is not busy, the OS/2 device driver may perform the requested activity immediately and return to the operating system. Otherwise, the device driver asynchronously notifies the operating system when the requested activity is completed. When the operation is complete, OS/2 returns to the requesting application. The requesting application can be a real-mode DOS application or a thread in a protect-mode OS/2 application process.

A device driver is the only component in OS/2 to interpret hardware interrupts.

Figure 3 shows the relationship between the OS/2 device driver and the system. API is the Application Programming Interface. The DOS interface is provided through Interrupt 21H (hexadecimal) and the OS/2 interface is provided through dynamically linked procedure-like calls to operating system services.

The role of a device driver in DOS is similar. DOS invokes the DOS device driver when the application issues a system request for I/O. The DOS application can also invoke the device driver through a software interrupt, provided that the device driver intercepts the software interrupt. But, in contrast to the OS/2 device driver, the DOS device driver performs the requested function and returns to its caller *only* when the requested action is complete.

The OS/2 device driver plays a more significant role in system operations than the DOS device driver. This coupling of the operating system and the device driver makes the OS/2 device driver appear as an extension of the operating system; an OS/2 device driver executes at the operating system privilege level, privilege level 0.

The role of an OS/2 device driver cannot be performed by an application, as an OS/2 device driver is the only component in OS/2 that is capable of interpreting hardware interrupts. Applications and I/O subsystems that manage a device must either use an existing device driver or supply their own device driver. This constraint results directly from the privilege-level architecture of the 80286 processor. A hardware interrupt handler must execute at the operating system privilege level. Applications and I/O subsystems, however, execute at the application privilege level, privilege level 3, or at the I/O privilege level (IOPL), privilege level 2.

OS/2 device management requires that the OS/2 device driver be bimodal, since mode switching to interpret application I/O requests would slow system response to application activity considerably. The OS/2 device driver must deal with application I/O requests and hardware interrupts regardless of the current mode of the processor.

Device drivers and application I/O requests

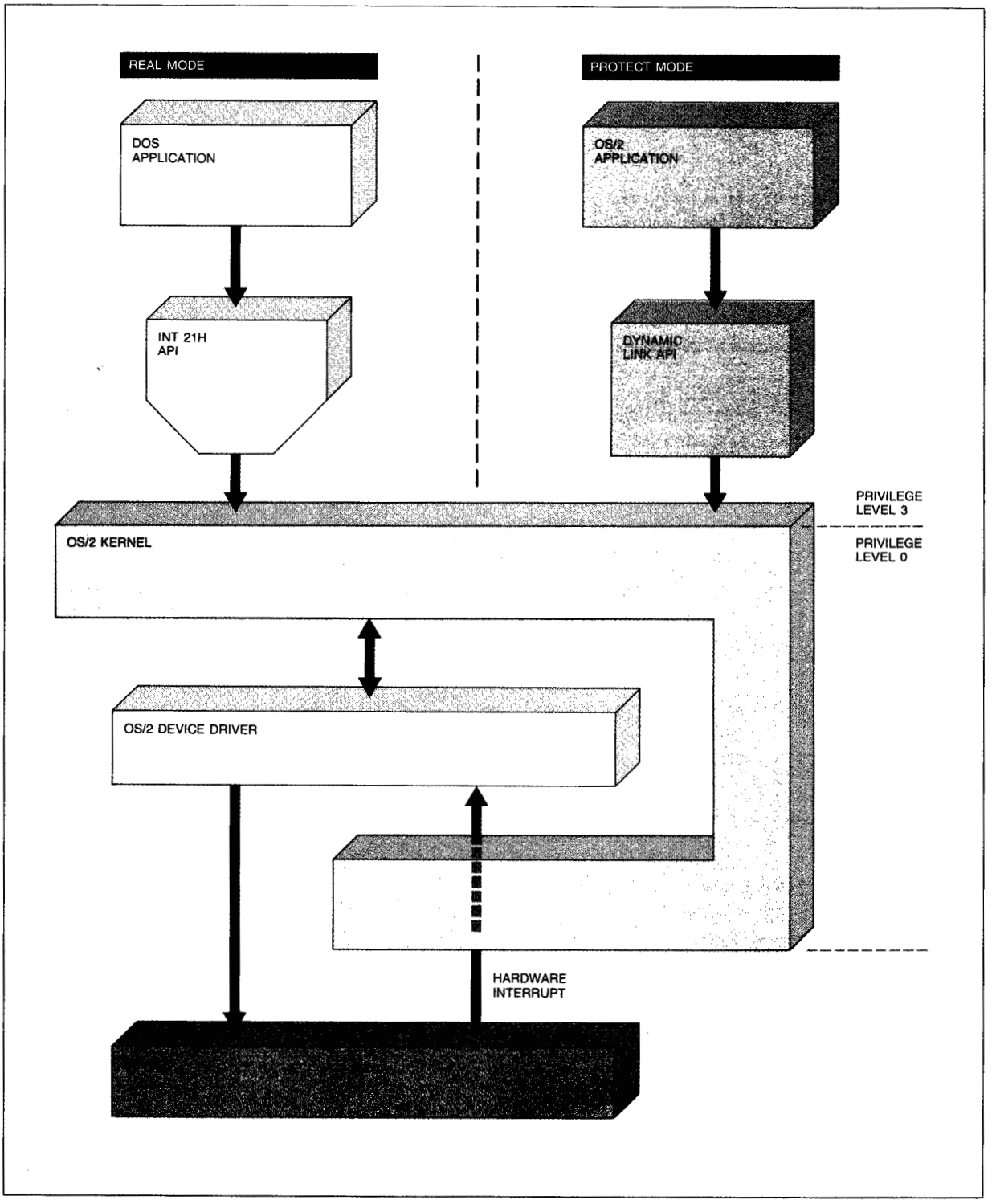
In OS/2, application requests for I/O are made to the operating system in the processor mode of the application. The operating system passes the request to the device driver, which then performs its operations under the context of the application. In protect mode, the device driver can address memory owned by the application through the application process's Local Descriptor Table (LDT). In addition, the device driver can address system memory through the Global Descriptor Table (GDT). In real mode, the device driver must use special system interfaces to address memory above 1M bytes.

Applications use different kinds of interfaces in OS/2 to perform I/O. The relationship between device driver request activity and application I/O requests depends on the system component that provides the particular application interface. The system component may send a number of requests to a device driver to satisfy a single application I/O request. In this situation, the device driver may not be able to extrapolate the original I/O request made by the application. If the commands received by the device driver correspond directly to the application interface, the device driver can identify the original I/O request. These scenarios depend on which set of rules the system applies to the use of the device.

Device models

As is done in DOS, OS/2 defines two device models, the block device model and the character device model, that determine how a device is treated by the

Figure 3 Relationship of the OS/2 device driver to the system



operating system and how an application may use the device. The choice of which device model to use for a particular device is made by the developer of the device driver. This choice depends on the nature of the supported device and on the application's intended use of the device. The device driver is then created according to the requirements of the model.

Block device model. A block device model is used for storage devices. These devices typically transfer data in blocks, usually through direct memory access (DMA). Data can be retrieved at any time and in any order; in other words, data transfer is not required to be done in the order in which the data appear on the device. A block device is accessed through the file system, which identifies a specific block device with a drive letter.

An application uses the file system interfaces to perform I/O to a file, which is an abstraction of block device data. The OS/2 File System defines a block of data as a sector 512 bytes long. The file system converts the application I/O requests to sector I/O commands to the device driver. The application performs I/O on the file by using the relative position of the data in the file in terms of bytes and specifying the number of bytes to transfer. The file system then maps the byte references of the application I/O request to the sectors on the device which correspond to the file. From a single application I/O request, the file system may send one or more sector-based I/O commands to the block device driver. The block device driver interprets I/O only in terms of sectors and in terms of sector position on the device; the device driver does not determine how data in a sector relate to the application request. The commands a block device driver can receive are listed in Table 1.

For example, an application uses the file I/O interfaces to first OPEN the file, READ and/or WRITE to the file, then CLOSE the file. The file system may respond to the OPEN by ordering the block device driver to READ sectors from the device which represent directory and other control information. The file system may also order the block device driver to WRITE sectors to the device in order to clear internal file system buffers. As a result of the application I/O (READs and WRITEs), the file system tells the block device driver to READ and WRITE sectors. For the CLOSE, the file system terminates the application's connection to the device. The file system may order the block device driver to WRITE sectors representing a final update to directory and other control information stored at the device. The block device driver never sees the exact type of request from the application; instead, the block device driver operates under the file system's I/O requests, which are in the context of application I/O requests.

Table 1 Block device driver commands

Initialize
Media Check
Build BIOS Parameter Block
Read
Write
Write With Verify
Removable Media Support
Generic IOCtl
Reset Media
Get Logical Drive Map
Set Logical Drive Map
Query Partitionable Fixed Disks
Get Fixed Disk/Logical Units

Another interface that can be used to access block devices is the I/O control interface, also known as the IOCtl interface. The IOCtl interface is used primarily to manage device-specific parameters that control how a device operates. As it applies to block devices, the IOCtl interface is generally used by system applications to format the media of a block device, not for file I/O. When a system application issues an IOCtl request, the operating system passes the IOCtl request directly to the target block device driver.

For example, a system application using the IOCtl interface first OPENs the device name with the file system interface. In this case, a block device name is a drive letter. The system application then issues IOCtl requests. When the system application has finished, it CLOSEs the device with the file system interface. The OPEN and CLOSE establish the connection of the system application to the block device. Each IOCtl request is sent to the block device driver identified by the application's connection to the device.

Character device model. A character device model is used for non-storage devices, which typically transfer data in terms of characters, usually through program I/O instructions (IN and OUT) to I/O ports. Data cannot be retrieved once they have been read or written. Here, the order of data is significant, because the device does not preserve the data.

A character device is accessed either through the file system, an I/O subsystem, or the device I/O control

Table 2 Character device driver commands

Initialize
Read
Peek (Nondestructive Read No Wait)
Input Status
Input Flush
Write
Output Status
Output Flush
Device Open
Device Close
Generic IOCtl
DeInstall

interface (IOCtl). A character device is identified to the system by a device name. Character device data may also be handled by an application through a character device monitor. The commands a character device driver can receive are listed in Table 2.

The primary file system interfaces used for character device I/O are the OPEN, CLOSE, READ, and WRITE functions. An application using the file system interfaces views character device data as a string of bytes or characters and performs I/O based on the relative position of the characters in the string and on the number of characters (bytes) to transfer. The OS/2 file system responds to application I/O requests by passing the requests to the character device driver without converting them.

A character device driver often manages an application's connection to its character device, particularly where there is no system component that controls access to the device. To control access to its device, a character device driver can choose to have the file system inform the device driver when an application establishes or terminates the connection to the character device. The character device driver uses a flag in its device header to indicate the choice of the device driver. Then, when the application OPENs the device, the file system sends an OPEN command to the device driver. When the application CLOSEs the device, the file system sends a CLOSE command to the device driver. The OPEN command allows the character device driver to associate subsequent I/O requests it receives (READs and WRITEs) with a particular application's connection. A character device driver may also initialize the state of the device whenever an application first makes the connection to the device (OPEN). Also, the device driver may allocate or initialize resources for the forthcoming application I/O. The CLOSE command allows the device driver to free any resources it has allocated on behalf of the application, reset internal variables, and shut down the device (if necessary).

Using file I/O interfaces, an application must first OPEN the character device name, READ and/or WRITE to the device, and finally CLOSE the device. The file system may respond to the OPEN by signaling the character device driver to establish the application's connection to the device. The file system responds to application I/O (READs and WRITEs) by sending READ and WRITE requests for bytes to the character device driver. For the CLOSE, the file system may respond by informing the character device driver to terminate the application's connection to the device. For a character device, the file system acts primarily to route the application I/O request to the character device driver. The character device driver can extrapolate the actual request made by the application.

The I/O control interface, or IOCtl interface, is used by an application to set device-specific parameters or functions. When an application makes an IOCtl request, the operating system sends the request directly to the target character device driver.

To use the IOCtl interface, an application first OPENs the character device name with the file system interface. It can then make IOCtl requests. When it has completed its use of the IOCtl interface, it CLOSEs the character device by using the file system interface. The OPEN and CLOSE are handled by the file system to inform the character device driver of the application's connection to the device. An IOCtl request is passed directly to the character device driver identified by the application's connection to the device.

I/O subsystem interfaces are used for device-specific I/O functions. OS/2 has three I/O subsystems: video (VIO), keyboard (KBD), and mouse (MOU). (The file system can also be perceived as an I/O subsystem for block devices.) An application views character device data in the logical format defined by the individual I/O subsystem. The VIO subsystem defines video data either as characters and display attributes or as a logical display buffer. Both the KBD and MOU subsystems define their respective data as records. When an application performs I/O with subsystem interfaces, the subsystem may use its own IOPL routines or some system interface(s) to accomplish the request. The system interfaces allow a subsystem to use a device driver as an application uses a device driver. The subsystem, however, may generate a number of requests in order to complete the activity requested by the application.

An application can directly manipulate character device data through the character device monitor interface. This interface allows an application to intercept the data stream going to or coming from a character device; i.e., an application can filter the device data by changing, inserting, or deleting characters in the data stream. The character device driver cooperates with the application by passing device data to the application. When the application is finished, it returns the data to the device driver, which then sends it to its ultimate destination. However, this interface applies only to those character device drivers which allow monitoring of their data streams, such as the base OS/2 keyboard, mouse, and printer device drivers.

Device driver structure

The structure of a device driver in OS/2, like that of a DOS device driver, follows the small program model. This means that the OS/2 device driver program has a single code segment and a single data segment. The OS/2 device driver program, however, does not include a stack segment. Instead, the device driver uses the stack of its calling program. A device driver performs its activities in a manner similar to a subroutine or procedure.

As with a device driver under DOS, the order of segments in the OS/2 device driver program is significant. The data segment must appear before the code segment. The file containing the device driver program is an OS/2-executable file (an EXE file) that contains, in the following order, the EXE header, the data segment, and the code segment. The EXE header is generated when the compiled program modules are linked together into a program file.

The data segment in the OS/2 device driver, as in a DOS device driver, must contain at least one device header. A device header is a control block that defines the device and the device model to the operating system. When the operating system loads the device driver program (during system initialization), the operating system examines the device header to determine whether the device driver is a block or character device driver. The operating system can then treat the device driver appropriately. For example, if the device driver is a block device driver, the operating system can assign drive letters to the block devices supported by the device driver. The device header also contains a pointer to the primary entry point into the device driver. The primary entry point is used by the operating system to communicate with the device driver. This entry point, which is called the Strategy Routine, is discussed later.

The code segment has one or more entry points which correspond to different device driver components.

Unlike a DOS device driver, the OS/2 device driver may contain extra data segments which may be used during the initialization of the device driver. However, after initialization, the extra data segments are discarded by the operating system. Typically, a device driver uses these extra segments to contain message text. These segments are bound to the executable (EXE) file containing the device driver program with the message utility MSGBIND. The messages are then accessed during device driver initialization through the message-handling interfaces in OS/2. The device driver can thus display status messages concerning its ability to initialize its device.

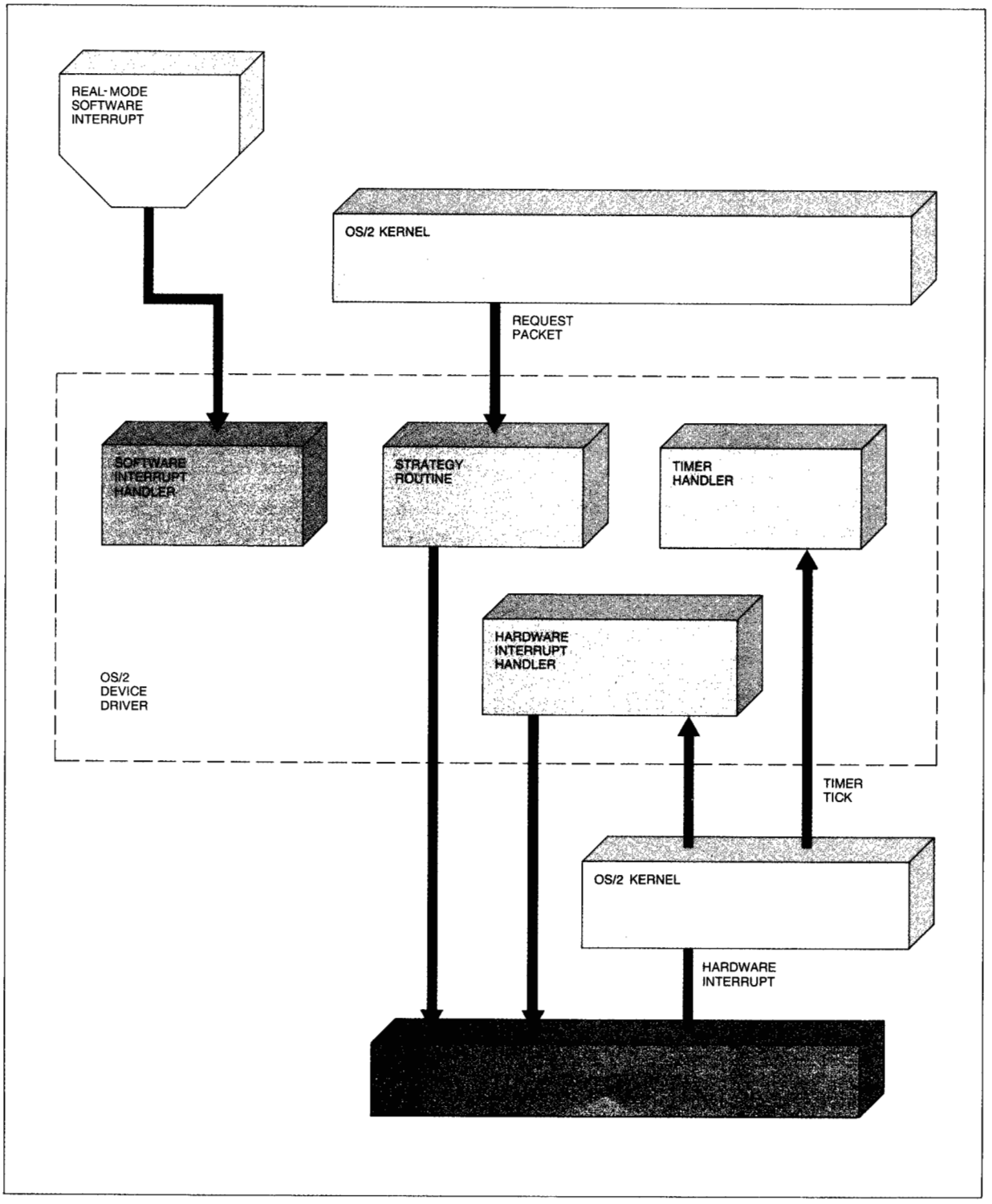
An OS/2 device driver may comprise up to four components: the Strategy Routine, the Hardware Interrupt Handler, the Timer Handler, and the Software Interrupt Handler (see Figure 4). These components interact with one another to control a device and to manage data transfer to and from the device. (A DOS device driver can have three types of entry points: the Strategy Routine, the Hardware Interrupt Handler, and the Software Interrupt Handler.)

Strategy Routine. The Strategy Routine is the fundamental component of the device driver; all device drivers have a Strategy Routine. In both DOS and OS/2, the Strategy Routine handles I/O requests on behalf of applications. Applications issue function calls to the operating system. The operating system converts function calls into commands to the device driver's Strategy Routine.

OS/2 has two application environments: OS/2 applications make I/O requests in protect mode; a DOS application in the DOS environment of OS/2 makes I/O requests in real mode. Consequently, the Strategy Routine of an OS/2 device driver is bimodal.

The interface between the Strategy Routine and OS/2 is the request packet. This interface is compatible with the interface used by DOS to communicate with DOS device drivers. The request packet is a control

Figure 4 Device driver components



block that contains the command to the device driver and command-related information. The device model determines which commands the device driver may receive (cf. Tables 1 and 2).

The Strategy Routine of an OS/2 device driver executes at task-time, that is, under the thread of the application process that made the system call. The application thread calls the operating system, which in turn calls the Strategy Routine. Because the Strategy Routine operates under the umbrella of the operating system, it is not subject to task switches as is an application. The thread executing in the Strategy Routine does not lose its execution until it blocks itself, yields its timeslice, or tries to access a segment that is not present in system memory. (A segment that is not present in system memory must be swapped from disk, during which time another thread can be dispatched.) For the device driver to maintain control over the device, immunity from preemption is important. The Strategy Routine can change the state of the device in order to handle the I/O command from the operating system. In order to guarantee control of the device, state changes must either be atomic or must run to completion. Consequently, the Strategy Routine controls its execution.

While executing at task-time, the Strategy Routine is subject to being interrupted by its device's hardware interrupts. If the Strategy Routine shares data structures with an interrupt-time component such as the Hardware Interrupt Handler, it must exercise care in accessing these common data structures.

To demonstrate how the Strategy Routine operates, an example of the activities that occur is outlined:

1. OS/2 calls the Strategy Routine entry point, passing a pointer to a request packet that identifies the activity the device driver must perform.
2. The Strategy Routine verifies the request packet.
3. If the command can be handled immediately (e.g., status check), the Strategy Routine performs the operation.
4. If the command requires activity by the device but the device is busy, the Strategy Routine places the request packet at the end of a work queue.
5. If the command requires activity by the device and the device is idle, the Strategy Routine issues the command to the device.
6. If the Strategy Routine has completed the requested function, it puts the status in the request packet and returns to its caller, the operating system kernel.
7. If the Strategy Routine has not completed the requested function, the Strategy Routine waits by blocking the thread. Blocking the thread releases the remainder of the time slice of the thread. The operating system will dispatch another thread, which can then make a system call. The Strategy Routine can be called again at its entry point.

Hardware Interrupt Handler. The Hardware Interrupt Handler is required if the device generates interrupts; all device drivers managing interrupting devices must have a Hardware Interrupt Handler. In OS/2, the device driver's Hardware Interrupt Handler is the only component in the system that may service a hardware interrupt. The Hardware Interrupt Handler is referred to as an interrupt-time component because it executes only when the device interrupt occurs. It executes as a special interrupt-time thread which is not associated with any application process.

The OS/2 device driver initializes the entry point to its Hardware Interrupt Handler during its initialization. When the hardware interrupt occurs, OS/2 calls the Hardware Interrupt Handler. In contrast, a DOS device driver can simply replace the appropriate vector in the real-mode interrupt table and let the device interrupt call the device driver directly.

Device activities are asynchronous to the activities being performed in the multitasking system. Consequently, device interrupts occur regardless of which application process is currently executing and regardless of which mode the processor is in. For example, an I/O request from a DOS application running in the foreground DOS environment may actually complete during the background execution of an OS/2 application. In this example, the I/O request is made in real mode, but the device interrupt signaling I/O completion occurs in protect mode. The reverse situation is also possible: an I/O request made in protect mode may have the corresponding device interrupt occur in real mode if the DOS environment is foreground. The result of this behavior is that the Hardware Interrupt Handler is bimodal, executing in either mode of the processor.

The actions of the Hardware Interrupt Handler may be illustrated by the following scenario:

1. The operating system calls the Hardware Interrupt Handler when the device interrupt occurs for which the Hardware Interrupt Handler is registered.

2. The Hardware Interrupt Handler confirms that its device issued a valid interrupt. A device may generate an interrupt because it has completed its work on a command or because it has experienced some spurious event unrelated to the command.
3. If its device issued the interrupt, the Hardware Interrupt Handler resets the interrupting condition at the device.
4. If the interrupt is not a spurious interrupt, the Hardware Interrupt Handler checks the current request packet to determine whether the requested function was completed.
5. If the requested function was completed, the Hardware Interrupt Handler puts the status in the request packet and "unblocks" the thread waiting in the Strategy Routine. The Hardware Interrupt Handler can also start the activity required for the next request packet in the work queue. In this case, the Strategy Routine can place packets on the queue and the Hardware Interrupt Handler can remove them.
6. If the requested function requires further work (a multistage operation), the Hardware Interrupt Handler tells the device to begin the next stage.
7. If the Hardware Interrupt Handler is done with activities for this particular occurrence of the device interrupt, it returns to its caller, the operating system kernel. The operating system will be able to call the Hardware Interrupt Handler at its entry point *at any time* after the Handler sends the the End-Of-Interrupt (EOI) to the hardware interrupt controller. The EOI tells the interrupt controller that pending device interrupts can be serviced. If there is another interrupt pending at the hardware interrupt controller for the Hardware Interrupt Handler, the Hardware Interrupt Handler will be called before the Handler actually returns to its caller for the first interrupt. This is known as "nesting" of interrupts.

Timer Handler. In an OS/2 device driver, the Timer Handler is an optional component. The OS/2 device driver can use a Timer Handler to manage timeouts and time delays. In this respect, the Timer Handler resembles the timing facility in the DOS environment. In the DOS environment, the timing facility is supplied by the Basic Input/Output System (BIOS) Int 1Ch interface. The OS/2 device driver can also use the Timer Handler to manage I/O operations for a noninterrupting device. In either case, the Timer Handler is actually driven by the system clock, the real-time hardware clock device, instead of the system timer that invokes the BIOS.

The OS/2 device driver initializes the entry point to the Timer Handler during its initialization. When the clock hardware interrupt occurs, OS/2 calls the device driver's Timer Handler. Like the Hardware Interrupt Handler, the Timer Handler executes at interrupt-time, and executes as a special interrupt-time thread. The Timer Handler is also bimodal, because the clock device generates interrupts regardless of the mode of the processor.

The OS/2 device driver uses the Software Interrupt Handler for software interrupts issued by the DOS application.

The equivalent kind of component in a DOS device driver is a software handler that intercepts the BIOS Int 1Ch interrupt or a hardware interrupt handler that interrupts the system timer.

The operations of the Timer Handler are shown in the following example.

1. The clock interrupt occurs, causing the operating system to call the Timer Handler.
2. The Timer Handler performs the activities appropriate for this time interval. If some action must take place in the Strategy Routine, the Timer Handler can "unblock" a thread that had been blocked by the Strategy Routine.
3. When done, the Timer Handler returns to its caller, the operating system kernel. The Timer Handler *does not issue an End-Of-Interrupt (EOI)* to the hardware interrupt controller. Instead, the clock device driver manages the clock interrupts.

Software Interrupt Handler. The Software Interrupt Handler is optional for OS/2 device drivers, which need it only to support operations in the DOS environment of OS/2. The OS/2 device driver uses the Software Interrupt Handler to intercept a software interrupt issued by the DOS application. On the other hand, a Software Interrupt Handler is a common

component for a DOS device driver. A DOS device driver often provides an interface to the DOS application through a software interrupt. In this situation, the DOS device driver sets the software interrupt vector to a handler in the device driver. The software interrupt therefore invokes the DOS device driver handler directly.

The OS/2 device driver must initialize the entry point of the Software Interrupt Handler for a specific software interrupt vector. Whenever the DOS environment is foreground, the DOS application may issue a software interrupt and directly invoke the device driver's Software Interrupt Handler. Since software interrupts occur only in the DOS environment, the Software Interrupt Handler executes only in real mode.

The OS/2 device driver's Software Interrupt Handler executes as an extension of the DOS application. In other words, the Software Interrupt Handler executes at task-time under the context of the DOS application, which means that the Software Interrupt Handler is subject to task switches. When the DOS application is foreground, it loses the processor for multitasking of background OS/2 application processes. In addition, the Software Interrupt Handler can be interrupted by hardware interrupts. It must therefore exercise caution whenever using data structures that are shared with other components in the OS/2 device driver.

The following example outlines the major activities of a Software Interrupt Handler that intercepts a BIOS software interrupt.

1. A DOS application issues the BIOS software interrupt, which invokes the Software Interrupt Handler.
2. The Software Interrupt Handler checks the requested BIOS function.
3. If the desired BIOS function is permitted and the device is not busy, the Software Interrupt Handler may either call the BIOS service to perform the function or perform the function itself.
4. If the desired BIOS function is permitted and the device is busy, the Software Interrupt Handler may set a flag to indicate BIOS I/O pending. The handler can then wait until one of the other device driver components indicates that the Software Interrupt Handler may proceed with the BIOS I/O.
5. When done, the Software Interrupt Handler performs an interrupt return (IRET) to the DOS application. The Software Interrupt Handler can be preempted by the multitasking of background protect-mode applications or can be suspended if the operator switches the DOS environment to the background. The Software Interrupt Handler can protect itself from suspension but not from preemption. It does this by issuing some special system calls to allow a critical section of execution to complete. However, the Software Interrupt Handler is not reentrant; it will not be called again until it returns to the DOS application.

Table 3 Rules for writing protect-mode code

Do not perform segment arithmetic with the segment registers.
Do not depend on "wrap-around" offsets.
Do not access beyond the end of a segment.
Put only valid selector values in the segment registers.
Do not write to a code segment.
Do not put both code and data in the same segment.
Do not depend on instruction execution speed.

Device driver operations

An OS/2 device driver performs its activities in the context of its caller and in the context of the processor mode. It interacts with the operating system to handle application I/O requests that originate from OS/2 applications and the DOS application in the DOS environment. The OS/2 device driver handles hardware interrupts that occur in protect mode and in real mode. It may even manage the access of a DOS application to a particular software interrupt interface such as BIOS. The combination of these activities leads to a number of requirements for an OS/2 device driver.

Bimodal considerations

Because the OS/2 device driver executes in protect mode and in real mode, the bimodal components of the device driver must follow rules enforced by the processor architecture for protect-mode execution (see Table 3). These rules also permit the device driver's bimodal components to execute in real mode.

System services

DOS has no services designed specifically for device-driver operations. OS/2, on the other hand, provides a set of special system services for device drivers, called helper routines, or DevHlp services. These services help the OS/2 device driver to interact with the operating system and to control the device driv-

Table 4 OS/2 DevHlp services by category

Process management
Semaphore management
Request Queue management
Character Queue management
Memory management
Interrupt management
Timer services
Character Monitor management
Advanced BIOS management

er's activities. Some system services provide functions commonly performed by device drivers. The availability of such services removes the need for each device driver to "roll its own" services.

OS/2 puts a pointer to the DevHlp interface in the Initialize request packet. When OS/2 calls the device-driver Strategy Routine to initialize, the device driver must save this pointer for later use. OS/2 sets up the pointer to the DevHlp interface so that the pointer is valid in both real mode and protect mode, and the device driver can call the DevHlp interface without considering which mode the processor is in. To request a DevHlp service, the OS/2 device driver places input parameters and the DevHlp function number into registers and makes a FAR CALL through the pointer to the DevHlp interface. The operating system performs the requested service depending on the availability of the service for the context in which the device driver is executing. The DevHlp services are listed by groups in Table 4.

Memory addressability

Bimodal operations make access to memory more complex. The device driver's interrupt-time components must take into account not only processor mode but also the current application context. An interrupt-time component will often execute under the address space (LDT) of an application different from the one which issued the I/O request. An interrupt-time component will often execute in a mode different from the mode of the I/O requestor if the system is configured for both OS/2 applications and a DOS environment.

The device driver utilizes the memory-management DevHlp services to help it manage its access to application memory. The device driver's task-time component, the Strategy Routine, uses a DevHlp service to convert memory addresses owned by an application process to physical addresses represented by 32-bit numbers. The Strategy Routine then saves the resulting 32-bit numbers. The device driver's interrupt-time components can convert the 32-bit numbers with a DevHlp service into temporary logical addresses. These temporary addresses give the device driver's interrupt-time components the ability to transfer data regardless of the processor mode or current application context.

An OS/2 device driver must also manage its access to nonsystem memory. (Nonsystem memory is the memory reserved by BIOS, and it covers the range of addresses from 640K bytes to 1M bytes.) OS/2 does not manage this reserved area of memory; to access this memory, an OS/2 device driver uses a DevHlp service to establish a temporary logical address. A device driver can then utilize memory such as the display buffers or memory that exists on an adapter.

Synchronizing component actions

The OS/2 device driver must manage its operations across its task-time and interrupt-time components. It may also need to coordinate activities with an application.

Synchronization among device driver components is relatively simple. Because all device-driver components have access to the device driver's data segment, the different components can share data structures. A common structure used in intercomponent communication is a work queue of request packets. Whenever its device is busy, the device driver can place requests in the queue. As a request is completed, it can be removed from the queue. All components of the device driver can therefore determine the current request. Another data structure is a RAM semaphore, which can be used to signal the occurrence of an event.

Another technique for intercomponent synchronization is the management of the thread in the Strategy Routine. The Strategy Routine can block the thread in order to wait for an interrupt event, and the interrupt-time components can unblock the thread when the event occurs.

For communication with an application, a device driver may use a system semaphore. In this case, the application passes the semaphore handle to the device driver, which converts the handle with a DevHlp service. The device driver uses other DevHlp services to manipulate the semaphore.

Interrupt management

There are several considerations for a device driver processing a hardware interrupt. Because of the sensitivity of the operating system to device-driver performance, the device driver must limit the time it disables interrupts. The device driver's interrupt-time components must also limit the total time spent processing the interrupt because time spent processing interrupts delays OS/2's response to multitasking events. The interrupt-time components must limit the amount of interrupt nesting that can occur, since interrupt nesting affects usage of the interrupt-time stack.

Interrupt nesting is the situation when the interrupt-time component is invoked before it finishes its current processing. In other words, nesting of interrupts is likely during the time between the sending of the End-Of-Interrupt (EOI) to the hardware interrupt controller and the exit from the interrupt-time component. Interrupt nesting depends on other factors as well, such as the interrupt rate of devices, the number of active devices, and the relative priorities of the active device interrupts. One technique the device drivers can use to limit interrupt nesting, assuming that the interrupt handler has no post-EOI processing to perform, is to disable interrupts prior to issuing the EOI. This prevents another interrupt from occurring before the current interrupt level completes, thereby reducing the possibility of a stack overflow.

Initialization

Initialization of an OS/2 device driver differs in a number of areas from that of other components in the system. A device driver is an installable component of the system. It is identified and loaded through the CONFIG.SYS configuration file, with the DEVICE command. The configuration file is processed by the operating system during system initialization to identify devices for subsequent I/O.

Although an OS/2 device driver is operationally bimodal, it initializes in protect mode. The context of initialization is a special operating system application-level process, sometimes referred to as the system initialization process. The device driver executes at the application protection level, privilege level 3, which allows the device driver to make dynamic link system calls. The device driver can therefore use the file system services to perform file I/O. For instance, the device driver can read a device configuration file or a font file. It can also use the message-handling facility to display messages. For example, the device driver may want to inform the user of the status of its attempt to initialize the device.

Although the device driver initializes while executing at the application-privilege level, the operating system grants it I/O privilege (IOPL). This permits the device driver to access I/O ports and disable hardware interrupts as necessary.

An OS/2 design point is compatibility with DOS applications.

Advanced BIOS

Advanced BIOS (ABIOS) is the device interface layer found on the Personal System/2™ Models 50, 60, and 80. It is important to the OS/2 device driver because it is a bimodal interface which uses request blocks to pass information. Without ABIOS, an OS/2 device driver must control its device directly, making the device driver sensitive to changes in the device. To help access the ABIOS, the device driver uses the DevHlp services provided by OS/2. OS/2 ensures that ABIOS is initialized prior to initializing the device drivers.

Support of DOS device drivers

One of the design points for OS/2 is compatibility with DOS applications. While many DOS applications can be supported with new OS/2 device drivers, it is also desirable to allow DOS device drivers to be installed in OS/2. However, the operating context of the DOS environment of OS/2 imposes some restrictions on the kinds of DOS device drivers that can be used in OS/2. The DOS device driver must be a character device driver; it must perform polled I/O; it must not have timing dependencies; and it must not rely on DOS Int 21H system calls during its initialization.

A DOS device driver executes only in the real mode of the processor. Under OS/2, a device managed by a DOS device driver can only be used by a DOS application running in the DOS environment. Because new OS/2 applications execute only in protect mode, they may not utilize the DOS device driver's device.

Summary

The OS/2 device driver is a key element in supporting the multitasking environment of OS/2. Interrupt-driven device management means that an OS/2 device driver can allow other activities to take place while waiting for completion of I/O to its device. An OS/2 device driver can also maintain a list of outstanding I/O requests. These factors help OS/2 to effectively utilize system resources.

General references

IBM Operating System/2 Technical Reference Volume I, 84X1434, IBM Corporation; available through IBM branch offices.

IBM Operating System/2 Technical Reference Volume II, 84X1440, IBM Corporation; available through IBM branch offices.

IBM Operating System/2 Programmer's Toolkit, Volume I, 6280200, IBM Corporation; available through IBM branch offices.

M. S. Kogan and F. L. Rawson III, "The Design of Operating System/2," *IBM Systems Journal* **27**, No. 2, 90–104 (1988, this issue).

iAPX 286 Programmer's Reference Manuals, Intel Corporation, Santa Clara, CA (1985).

iAPX 286 Operating System Writer's Guide, Intel Corporation, Santa Clara, CA (1983).

Ann M. Mizell *IBM Entry Systems Division, 1000 NW 51st Street, Boca Raton, Florida 33432.* Ms. Mizell joined IBM in 1982 in Boca Raton as an application developer for the IBM Personal Computer. She is currently an advisory programmer in the systems software architecture group, where she has participated in the design and development of OS/2. Ms. Mizell received her B.S. degree in systems and controls engineering from Case Western Reserve University, Cleveland, Ohio, in 1982; she is a member of the Tau Beta Pi engineering honor society. Ms. Mizell is co-author of a book entitled *OS/2 Features, Functions, and Applications*, John Wiley & Sons, Inc., New York (1988).

Reprint Order No. G321-5317.