Microsoft Operating System/2

Device Drivers Guide

Microsoft Corporation

Document Number 510830010-100-O00-0487
Part Number 110-098-022

Contents

Figures

Tables

How to Use This Manual

Introduction

The Microsoft® Operating System/2 manual set describes MS OS/2, a multitasking operating system for the Intel 80286 microprocessor. Three manuals in the set discuss the operating system, and are intended for the novice user:

- *Microsoft Operating System/2 Setup Guide*
- *Microsoft Operating System/2 Beginning User's Guide*
- *Microsoft Operating System/2 User's Reference*

The other three manuals discuss topics of interest to programmers:

- *Microsoft Operating System/2 Programmer's Guide*
  A tutorial introduction to MS OS/2
- *Microsoft Operating System/2 Programmer's Reference*
  A reference for all MS OS/2 functions
- *Microsoft Operating System/2 Device Drivers Guide* (this manual)
  A reference for the MS OS/2 device drivers

Manual Purpose

The *Microsoft Operating System/2 Device Driver's Guide* describes the architecture and operating details of MS OS/2 device drivers. It includes detailed reference pages for device driver commands, Generic IOCtl functions, and device helper services.

Manual Organization

The *Microsoft Operating System/2 Device Driver's Guide* is organized to help you find the information you need to understand the operation of device drivers for MS OS/2. The Guide is divided into front matter and five chapters.

Front Matter

The front matter in this manual contains a Table of Contents describing the major headings of each chapter and section. In addition to the main Table of Contents, each chapter begins with a Table of Contents.

The front matter in this manual also contains this chapter on the purpose of the manual and suggestions on how to find the information contained within it.

The Chapters

The parts and chapters in the *Microsoft Operating System/2 Device Driver's Guide* are listed and summarized as follows:

Chapter 1, "Device Driver Architecture" provides an architectural overview of MS OS/2 device drivers. It also discusses the device driver interface to MS OS/2.

Chapter 2, "Device Driver Descriptions," describes each of the major types of device drivers and presents details and restrictions on their usage in real and protected modes.

Chapter 3, "Device Driver Commands," describes the structure of the device driver header for MS OS/2 and the device driver request packet structure. It details each of the MS OS/2 device driver command codes.

Chapter 4, "Generic IOCtl Commands," details each of the Generic IOCtl functions for each device driver category supported by MS OS/2.

Chapter 5, "Device Helper Services," details each of the MS OS/2 device helper (**DevHlp**) functions. They are listed in categories according to their functionality.

Appendix A, "Translate Table Format," details the structure of the MS OS/2 keystroke translation table.

Notational Conventions

Throughout this manual, the following conventions are used to distinguish elements of text:

bold	Bold type is used for commands, options, switches, and literal portions of syntax that must appear exactly as shown.
italic	In addition to giving emphasis to new terms, italics are used for filenames, variables, and placeholders that represent the type of text to be entered by the user.
`monospace`	Monospace type is used for sample command lines, program code and examples, and sample sessions.
SMALL CAPS	Small caps are used for keys, key sequences, and acronyms.

Special Characters and Symbols

The MS OS/2 operating system requires that you use, and be familiar with, many special characters and symbols. For simplicity and consistency, the MS OS/2 manuals use the following set of characters and symbols:

Symbol	Name	Function
&	ampersand	command separator
&&	AND operator	logical AND
*	asterisk	wildcard character
@	at symbol	used in code examples to indicate FAR addresses
\	backslash	path and filename separator
[]	brackets	enclose optional syntax items
^	caret	escape character

$	dollar sign	used in device names
>	greater-than sign	redirection symbol
<	less-than sign	redirection symbol
≠	not-equal sign	arithmetic symbol
!=	not-equal sign	used in code examples
#	number sign	used in code examples
||	OR operator	logical OR
()	parentheses	command grouper
%	percent sign	used for replaceable parameters
|	pipe symbol	redirection symbol
±	plus/minus sign	arithmetic symbol
?	question mark	wildcard character
>>	redirection symbol	used to append output
/	slash	used in switches

Chapter 1
Device Driver Architecture

1.1 Overview

A *device driver* is a program responsible for communication between MS OS/2 and the system hardware. When an application program requests device I/O, the device driver acts as a software interface to perform the I/O request on behalf of the operating system. This request is queued internally until the device is free to process it.

In a single-tasking operating system such as previous versions of MS-DOS, it was acceptable to use synchronous, non-interrupt-driven device drivers. This was because program execution could not proceed until device I/O was complete.

MS OS/2, however, is a multitasking operating system. It is important that MS OS/2 device drivers are interrupt-driven and that they surrender the CPU while they are waiting for I/O completion. The operating system can then assign the CPU to other tasks that are not waiting on I/O.

In addition, MS OS/2 device drivers must provide support for compatibility-mode operations, which are a set of old applications that run in real mode. This means that MS OS/2 device drivers may need to interlock real-mode ROM BIOS device I/O with protected-mode device I/O, as well as to handle I/O requests from real-mode applications. MS OS/2 device drivers must also be able to handle hardware interrupts without the overhead that results when switching modes. They must be *bimodal*; that is, they must be able to execute in both protected mode and real mode.

The basic characteristics of the MS OS/2 device driver, then, are the support for the multitasking environment, and the ability to execute in both real and protected mode.

1.2 Types of Device Drivers

There are two types of device drivers: *character* device drivers and *block* device drivers.

Character device drivers manage I/O on character-oriented devices, which perform I/O on a character-by-character basis. A character device driver (which has a name like *SCREEN*$, *KBD*$, or *PRN*), may support more than one device by having multiple, linked device driver headers, with

each header indicating a different name.

Block device drivers support block-oriented devices, which perform I/O on a block of data, typically through *Dynamic Memory Allocation* (DMA). In contrast to character device drivers, block device drivers do not have names, because on block-oriented devices, applications request I/O by means of a file system. Instead of a name, a block device driver is assigned drive letters, one for each unit (or device) supported by the block device driver, which can support multiple devices.

A block device driver specifies the number of devices that it supports when it is initialized (for more information about initialization, see the **Init** command in Section 1.11, "Device Driver Initialization").

The position of the device driver in the chain of all device drivers determines the way the drive letters correspond to the physical block devices (units). The order in which any installable block device drivers appear in the *config.sys* file (in the **device**=*command*) determines the order in which they are assigned drive letters.

1.3 Device Driver Model

The MS OS/2 device driver is a *small model*, meaning it consists of two segments: a code segment and a data segment. The device driver code must not have internal references to its own code segment selector (this means no FAR CALLs or FAR JUMPs). Only MS OS/2, at both task and interrupt time, has to know the code segment or selector; it keeps track of both values and uses the one appropriate to the current mode. Similarly, MS OS/2 tracks the data segment/selector and sets the data segment register (DS) to the value appropriate for the current mode.

1.4 Components of a Device Driver

MS OS/2 device drivers support multiple synchronous and asynchronous I/O requests. For example, a disk device driver can queue several read and write requests from multiple requesters, and service those requests in an order that minimizes any movement of the access mechanism across the disk.

MS OS/2 device drivers have the following parts:

- A *strategy routine*

  This routine is called to handle I/O requests with the MS OS/2 kernel. It must be able to run in both real and protected modes.

  By convention, the strategy routine need not save and restore any registers used, since the kernel does this before calling the driver.

  The strategy routine, which is called with an I/O packet that describes a request, marks the request incomplete and then queues it. If the device is not busy, it starts the device, and then returns to the kernel, which typically blocks the incomplete I/O packet.

- A *hardware interrupt routine*

  This routine is called as the result of a hardware interrupt. The interrupt routine must follow the FAR CALL/RET model, so when processing is complete, it must perform a FAR RET. The interrupt routine must be able to run in both real and protected modes.

  The interrupt routine must save and restore any registers used, except for flags. Once it has completed processing for its interrupt, it must clear the carry flag (CF = 0) prior to the FAR RET.

  The interrupt routine services the I/O completion. If there is new work in the queue, it redrives the device. It then indicates that the previous operation is complete and unblocks any threads that are waiting for the request to complete.

  The interrupt routine should run (as much as possible) with interrupts enabled. In addition, it should dismiss the interrupt at the 8259 interrupt controller as soon as possible.

- *ROM BIOS compatibility support routines* (optional)

  Some device drivers need to intercept ROM BIOS software interrupts from the compatibility-mode box. These ROM BIOS support routines run only in real mode.

- *Device helper routines* (optional)

  In addition to the strategy and interrupt routines, MS OS/2 provides device driver helper routines for, among other things, managing the request queue, blocking and unblocking processes, and locking and unlocking memory. For more information about these routines, see Chapter 5, "Device Helper Services."

1.5 Device Driver Modes

MS OS/2 device drivers operate in four modes:

- *Kernel Mode*

  When calling the device driver strategy routine, the MS OS/2 kernel uses Kernel mode, which applies to both real and protected mode. The MS OS/2 kernel calls the strategy routine and passes it a request packet containing information about the request. The strategy routine will not be preempted by a task switch but may be interrupted by incoming hardware interrupts. For more information about the commands and the associated request packet formats, see Chapter 3, "Device Driver Commands."

- *Interrupt Mode*

  When calling the device driver interrupt routine, MS OS/2 uses Interrupt mode, which applies to both real and protected mode. The interrupt routine is invoked when it receives a hardware interrupt.

- *User Mode*

  When calling the device driver ROM BIOS interrupt handler, MS OS/2 uses User mode, which applies only to real mode. The ROM BIOS interrupt handler is invoked by a ROM BIOS software interrupt. In this mode, the ROM BIOS interrupt handler may be preempted by a task switch.

- *Initialization Mode*

  When calling the device driver strategy routine, MS OS/2 uses Initialization mode, which runs only in protected mode. To call the strategy routine, MS OS/2 uses a request packet containing the **Init** command. The initialization code runs (in protected mode) at the application privilege level with I/O privilege. A limited set of dynamic-link system calls are available for use, as well as a portion of the device helper (**DevHlp**) function calls. For more information about Init mode and device driver initialization, see Section 1.11, "Device Driver Initialization."

Two additional terms must be defined: *task time* and *interrupt time.* Task time is a generic term that refers to executing code as a thread within a process. Interrupt time is a generic term that refers to executing code as a result of an interrupt; the "thread" of execution does not belong to a

process.

1.6 Bimodal Device Driver Operations

The following scenario demonstrates the interaction between the strategy routine and the interrupt routine of a device driver:

- The handling of an I/O request begins with MS OS/2 passing a request packet to the strategy routine when it calls the entry point of the device driver.
- The strategy routine then checks the validity of the I/O request. If the request is valid, the strategy routine places it on a work queue for the device, using the **DevHlp** functions to manage the request queue. But if the device is currently idle, the strategy routine starts it and then returns to MS OS/2, which typically suspends the thread until the I/O request has been completed.
- When the device interrupt occurs, the interrupt routine must set the return status in the request packet, remove the request packet from the queue, and call the **DevHlp** function **DevDone** to indicate to MS OS/2 that the I/O request is complete. If the device is ready for the next request, the interrupt routine starts it then exits by clearing the carry flag and doing a FAR RET.

Note

The strategy routine queues the requests and only initiates the I/O if the device had been inactive; the interrupt routine, however, starts requests as they reach the head of the work queue. In addition, to avoid races with the interrupt routine, the strategy routine must lock out interrupts when determining whether the device can be started.

The task time portions of a device driver should be fully reentrant and capable of processing multiple requests simultaneously. MS OS/2 will not preempt a thread in a device driver; instead, context is switched only when the driver issues a **Block**, **Yield** (**TCYield**), or references a movable/swappable segment. Once the strategy routine relinquishes the

context of a request (from issuing a **Block**, **Yield** (**TCYield**) or referencing a movable/swappable segment), the strategy routine may be called with a new request under the context of a different thread. Note that the strategy routine's code and the device driver data segment are the same across these invocations of the strategy routine.

The strategy routine can assume that it will not be preempted by other task time instances, but it must protect itself against its own interrupt routine. To protect itself, the strategy routine disables interrupts when checking whether the device is active, and when examining the device queue. Then the strategy routine will be preempted only by other higher priority interrupts, and only if it reenables interrupts.

The context in which a device driver is issued a request will not necessarily be the same context in which it is notified of the request's completion. For example, the address of a user buffer passed to the driver when a request is issued might refer to *Local Descriptor Table* (LDT) other than the current LDT when the request completes. This requires the device driver to store buffer addresses as 32-bit physical addresses.

Figure 1.1 shows how device drivers queue and process requests.

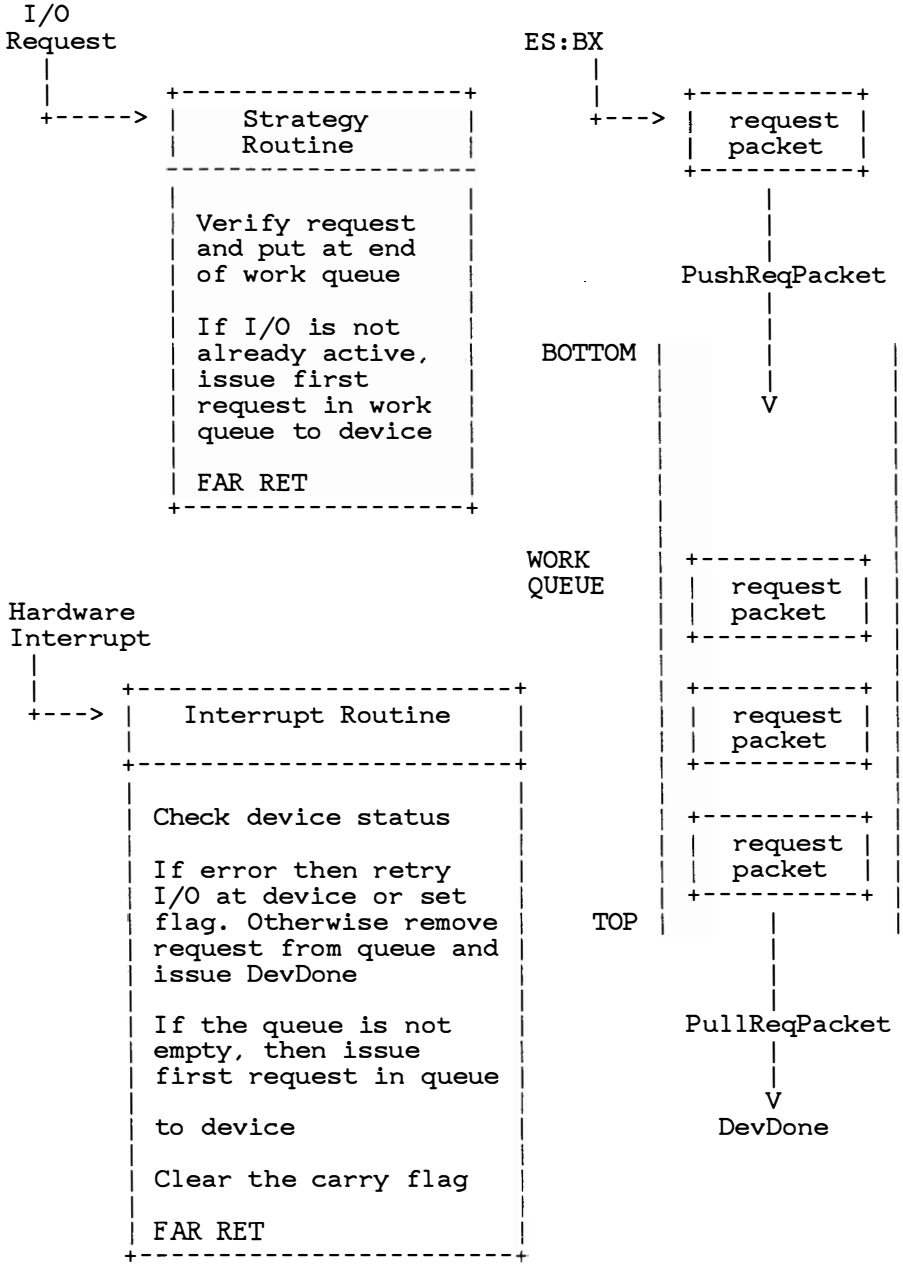

I/O
Request
Strategy
Routine
Verify request
and put at end
of work queue
If I/O is not
already active,
issue first
request in work
queue to device
FAR RET
ES:BX
request
packet
PushReqPacket
BOTTOM
V
WORK
QUEUE
request
packet
request
packet
request
packet
TOP
PullReqPacket
V
DevDone
Hardware
Interrupt
Interrupt Routine
Check device status
If error then retry
I/O at device or set
flag. Otherwise remove
request from queue and
issue DevDone
If the queue is not
empty, then issue
first request in queue
to device
Clear the carry flag
FAR RET

Figure 1.1 MS OS/2 Device Driver Model

1.7 Compatibility Support of Real-Mode I/O

A device driver is responsible for managing I/O for its device. Since many devices must be accessible from both real and protected mode (such as disk drive, keyboard, screen, and mouse devices), a MS OS/2 device driver must manage its device across both modes of operation.

Device I/O in real mode is performed in one of three ways:

- through the DOS INT 21H interface
- through the ROM BIOS interface
- through direct access to the device

I/O sent via DOS INT 21H is transformed into the request packet interface, which the device driver receives. (Similarly, I/O via MS OS/2 dynamic-link function calls is transformed into the request packet interface.) Since the request packet interface is standard across modes, the new device driver has no problem managing its device.

I/O sent via ROM BIOS poses some problems for a MS OS/2 device driver. The MS OS/2 device driver must intercept the ROM BIOS software interrupt (by setting the vector using the **DevHlp** function **SetROMVector**), and interlock ROM BIOS operations on its device in two ways:

- Serialize access to the device.

  Serialization can be performed by using semaphores to indicate when the device is busy with a request (and consequently cannot accept/tolerate a request from ROM BIOS).

- Protect critical sections of ROM BIOS execution from being blocked (frozen).

  Sections of ROM BIOS code must be protected from being frozen. When a user switches away from the real-mode session, the real-mode session box becomes suspended in the background. However, some I/O processing, such as the printer, (BIOS INT 17H), disk (BIOS INT 13H), and screen (BIOS INT 10H), cannot tolerate being suspended. It is the responsibility of the MS OS/2 device driver to intercept the appropriate ROM BIOS interrupt and issue the **DevHlp** function, **ROMCritSection** to protect the ROM BIOS critical section from executing.

Warning

When the MS OS/2 device driver issues **ROMCritSection** to "enter" a ROM BIOS critical section, you will not be able to switch from the screen of the real-mode session to another screen. This may cause problems. For example, if a real-mode terminate-and-stay-resident program takes control while the CPU is executing the ROM BIOS, the time spent in the ROM BIOS critical section will be longer, and you will be unable to switch screens. The worst case would be a terminate-and-stay-resident application that is interactive. It would never allow the MS OS/2 device driver to issue the exit from the critical section, and you would not be able to switch from the screen of the real-mode session until you terminate the application.

Application I/O via direct access to a device driver's device poses the same problem for the MS OS/2 device driver under MS OS/2 as it does to an MS-DOS device driver operating under MS-DOS. If device-state information is critical or if device I/O must be serialized, a device driver may choose not to access the full function of the device; control of the device, in this case, would be shared between the application and the device driver.

1.8 Request Queue Management

The strategy routine may either queue a request block or process it immediately. Typically, only read and write requests need to be queued; other types of requests can usually be handled immediately by the strategy routine.

A block device driver, such as the disk device driver, may process queued requests in any order. For instance, to shorten device access time, the block device driver may choose to sort the requests. A character-oriented device driver should always handle queued requests in the order it received them; otherwise, mixed output could result.

The device driver may manage its queues with the MS OS/2 **DevHlp** functions. In such a case, the device driver must allocate a *queue head* (a DWORD data area that will contain a pointer to the first request packet in the queue) and initialize it to zero. The device driver must set up a queue head for each queue it uses.

MS OS/2 calls the strategy routine with a bimodal pointer to the request packet; this pointer is valid in both real and protected mode. Any addresses passed in the request packets for read/write requests are passed as 32-bit physical addresses (normalized). Therefore, the device driver need not lock or convert the addresses into physical addresses. Instead, it needs to lock only addresses that it receives from a source other than MS OS/2, such as from a process passing an address via a Generic IOCtl call.

1.9 Sharing Hardware Interrupts

A hardware interrupt level (IRQ) that is shared among two or more devices is referred to as a *shared interrupt.* A single device driver with an interrupt handler for a particular interrupt level may manage the sharing of that interrupt among its own devices. Shared interrupts can only be used with devices of similar or identical nature; for example, a printer device driver supporting several printers on one interrupt level.

However, the model of a single device driver managing multiple devices on one interrupt level is not the typical case of interrupt sharing. The usual case is to have two or more devices of diverse nature, each managed by its own device driver, sharing a particular interrupt level.

For the IBM PC/AT, interrupt level 7 (IRQ7) is defined to be a shareable interrupt level.

To permit two or more device drivers to share an interrupt level, each device driver must adhere to the following rules:

- The device driver must indicate when signing up for an interrupt level via the **SetIRQ** call that it will share the interrupt level.

  If a device driver indicates that it cannot share the interrupt level, any subsequent device driver that wants to share that interrupt level will not be allowed to sign up for it. Conversely, if a device driver signs up as sharing the interrupt level, any subsequent device driver that wants exclusive access to that interrupt level will not be allowed to sign up for it.

Note

The interrupt sharing mechanism is defined only for the MS OS/2 bimodal device drivers. A hardware interrupt will not be shared among a mixture of interrupt handlers from MS OS/2 bimodal device drivers and MS-DOS real-mode-only device drivers.

- The device driver interrupt handler, when invoked, must always check to see whether its device owns the interrupt or not.

  To improve performance, if the interrupt handler owns the interrupt, it should enable interrupts and issue the *End-Of-Interrupt* (EOI) as soon as possible. If the interrupt handler does not own the interrupt, it must not enable interrupts, nor should it issue the EOI.

- After taking the appropriate action in processing the interrupt, the interrupt handler must indicate whether or not it claimed the interrupt.

  If the interrupt handler owns the interrupt, it must clear the carry flag (CF = 0) and issue a FAR RET when processing is complete. If the interrupt handler does not own the interrupt, it must set the carry flag (CF = 1) and issue a FAR RET.

1.10 Asynchronous and Synchronous I/O

In MS OS/2, asynchronous I/O is performed by using a separate thread. When a read or write request is made indicating that asynchronous processing is desired, the kernel device support creates an asynchronous thread. This thread then performs the I/O request synchronously (to itself), while the requesting thread continues executing asynchronously. When the operation is complete, the I/O thread posts that completion to a RAM semaphore and then terminates. When asynchronous I/O is used in this manner, it is transparent to a device driver whether an I/O request is synchronous or asynchronous.

Figure 1.2 depicts the flow of control for synchronous I/O:

- The strategy routine marks the request incomplete and queues the request.
- If the device is not busy, the strategy routine starts it.
- Until the interrupt routine indicates the request is done, execution of the thread is blocked in the kernel.

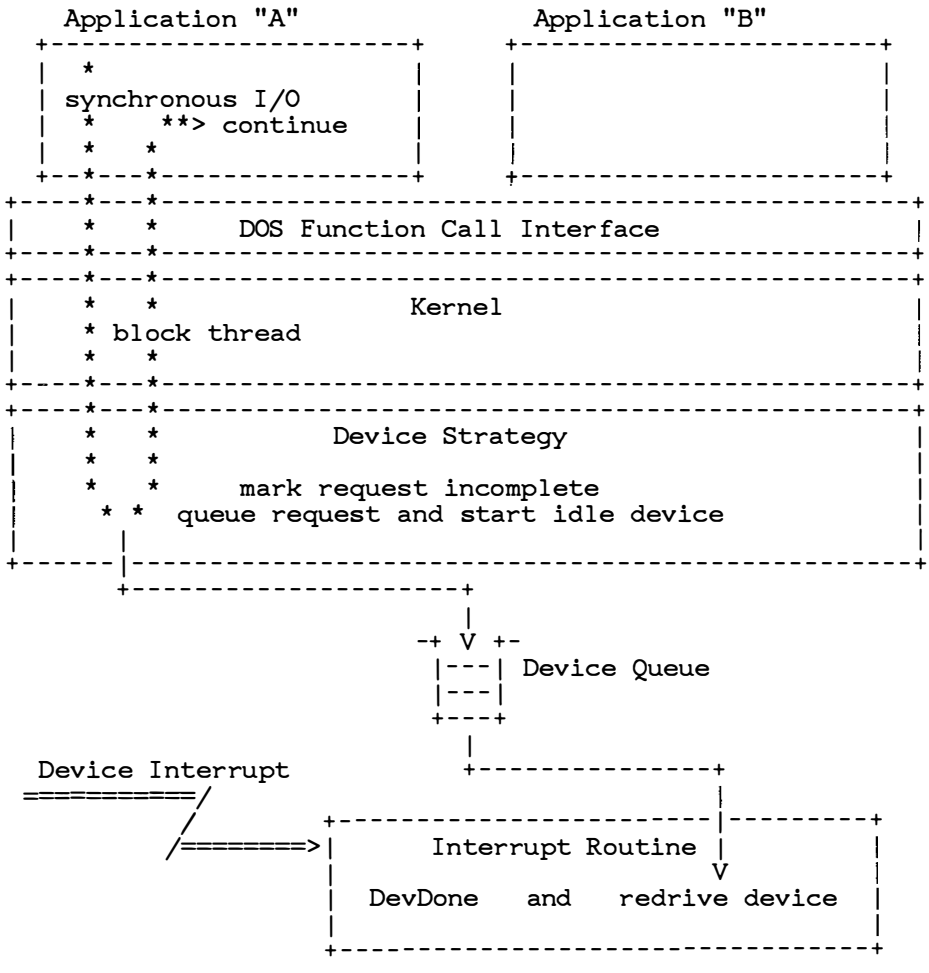


Figure 1.2 Device Driver—Synchronous I/O

Figure 1.3 depicts synchronous and asynchronous I/O. The flow is similar to the synchronous I/O depicted in Figure 1.2, except that the requesting thread continues to execute while a separate I/O thread executes the I/O.

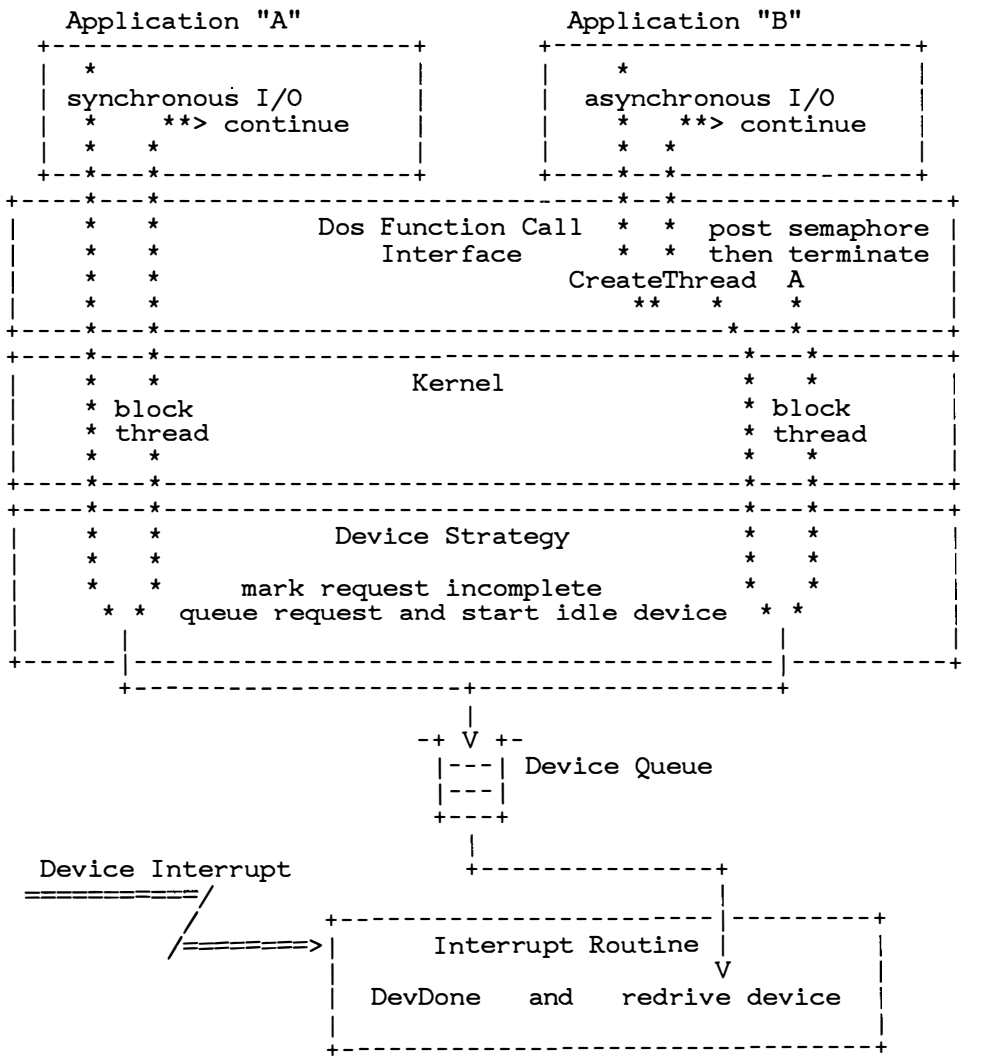


Figure 1.3 Device Driver—Synchronous and Asynchronous I/O

1.11 Application/Device Driver Interface

New MS OS/2 applications access device drivers through two basic interfaces:

- The **DosDevIOCtl** system call
- The file I/O system calls

The **DosDevIOCtl** system call is used to access both character and block device drivers. The application must first issue a **DosOpen** on the device name in order to obtain a device handle that identifies the target device.

When a **DosDevIOCtl** is issued, the device driver is called with a request packet in the Generic IOCtl format. For more information about the format of this request packet, see Chapter 3, "Generic IOCtl: I/O Control for Devices (Command Code 16)."

The file I/O systems calls are used primarily to perform I/O on *Direct Access Storage Devices* (DASDs) or block devices. Character device drivers, however, may also be accessed by the **DosOpen**, **DosClose**, **DosRead**, **DosReadAsync**, **DosWrite**, and **DosWriteAsync** system calls.

1.11.1 Console Device Interface Management

You can send I/O to and from the console by handle-based I/O, which can be redirected, or by a *Console Device Interface* (CDI). Figure 1.4 depicts handle-based console I/O.

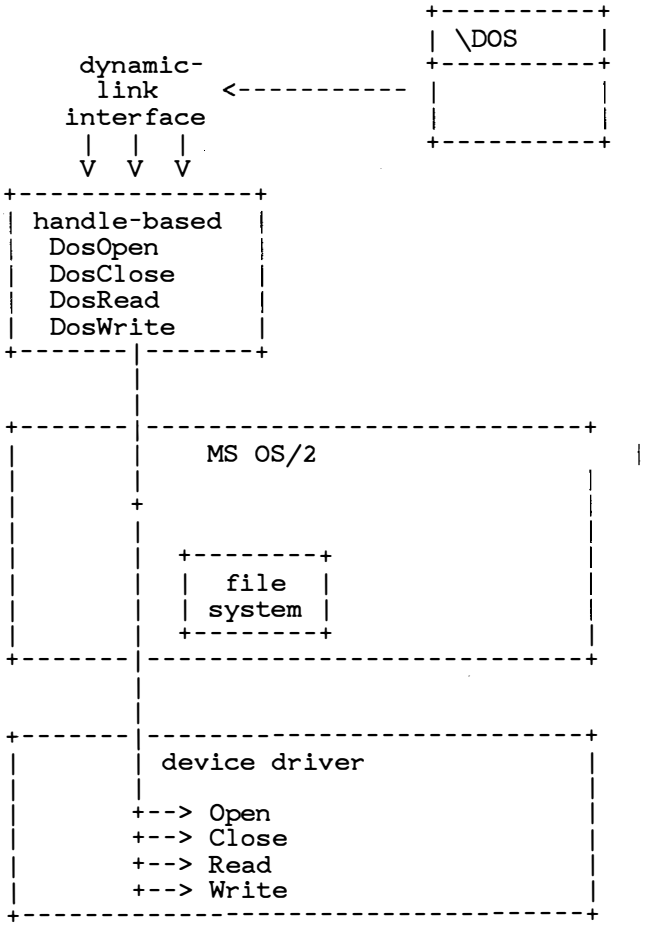


Figure 1.4 Handle-based Console I/O

Figure 1.5 shows console I/O that is not redirectable, such as direct cursor-positioned I/O provided by a dynamic-link CDI:

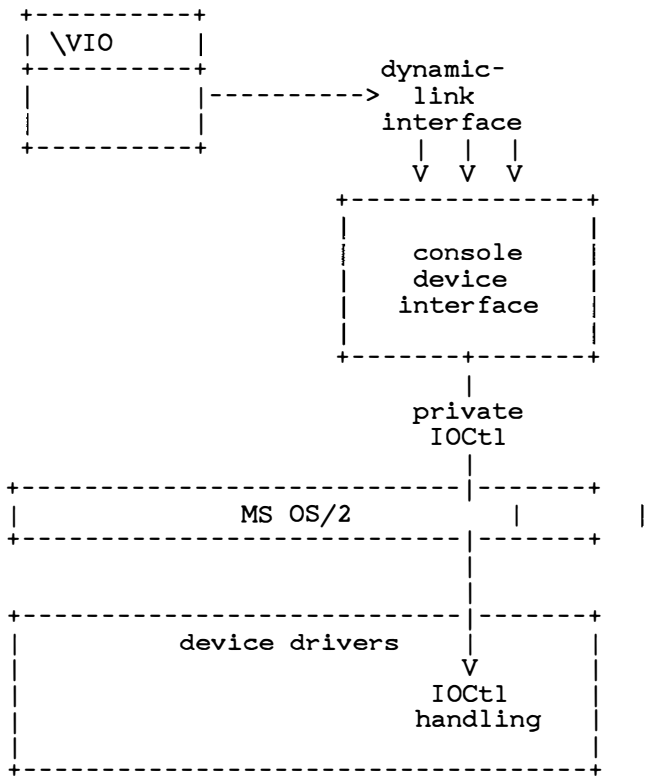


Figure 1.5 Console Device Interface

The console device interface and its device drivers are pairs. A private protocol IOCtl provides flexible function call placement and access to a variety of hardware features.

Figure 1.6 shows how a function is split between a console device interface and its device drivers, (which access the display hardware). In this case, the console device interface updates and manages the logical display buffers, even though the device drivers could manage them, too.

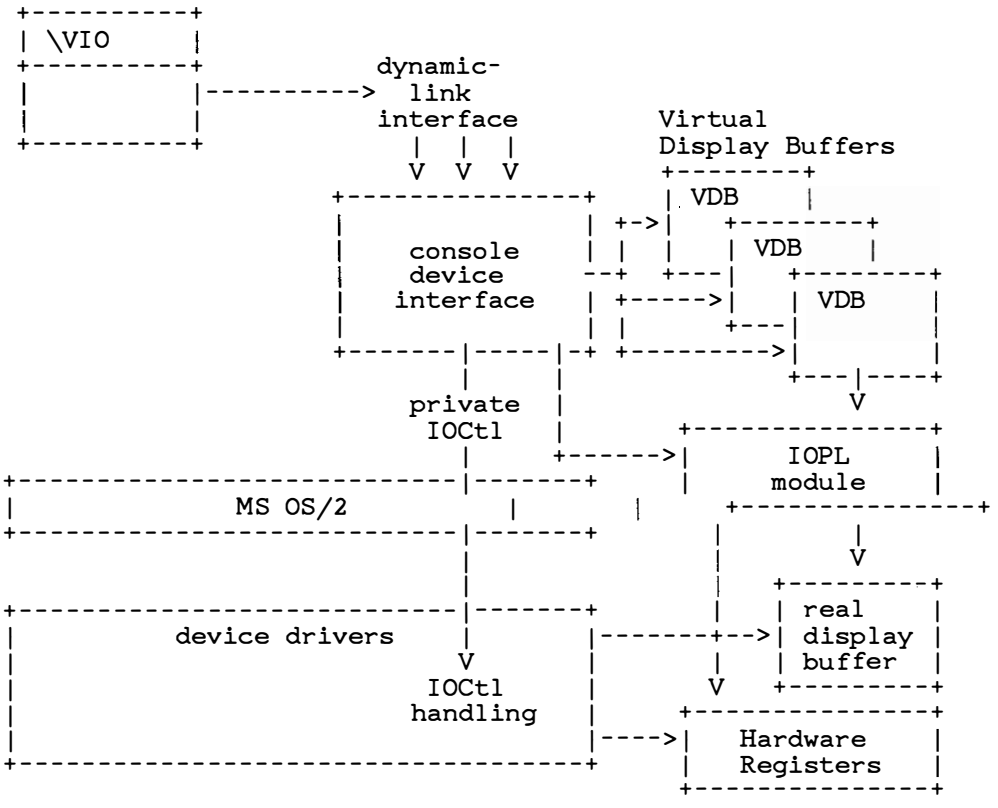


Figure 1.6 CDI Device Driver Manages Hardware

Only the device drivers and registered IOPL modules have *I/O Privilege Level* (IOPL).

Handle-based console I/O must written to the virtual display buffer before it is mapped to the real display buffer. Figure 1.7 illustrates this process.

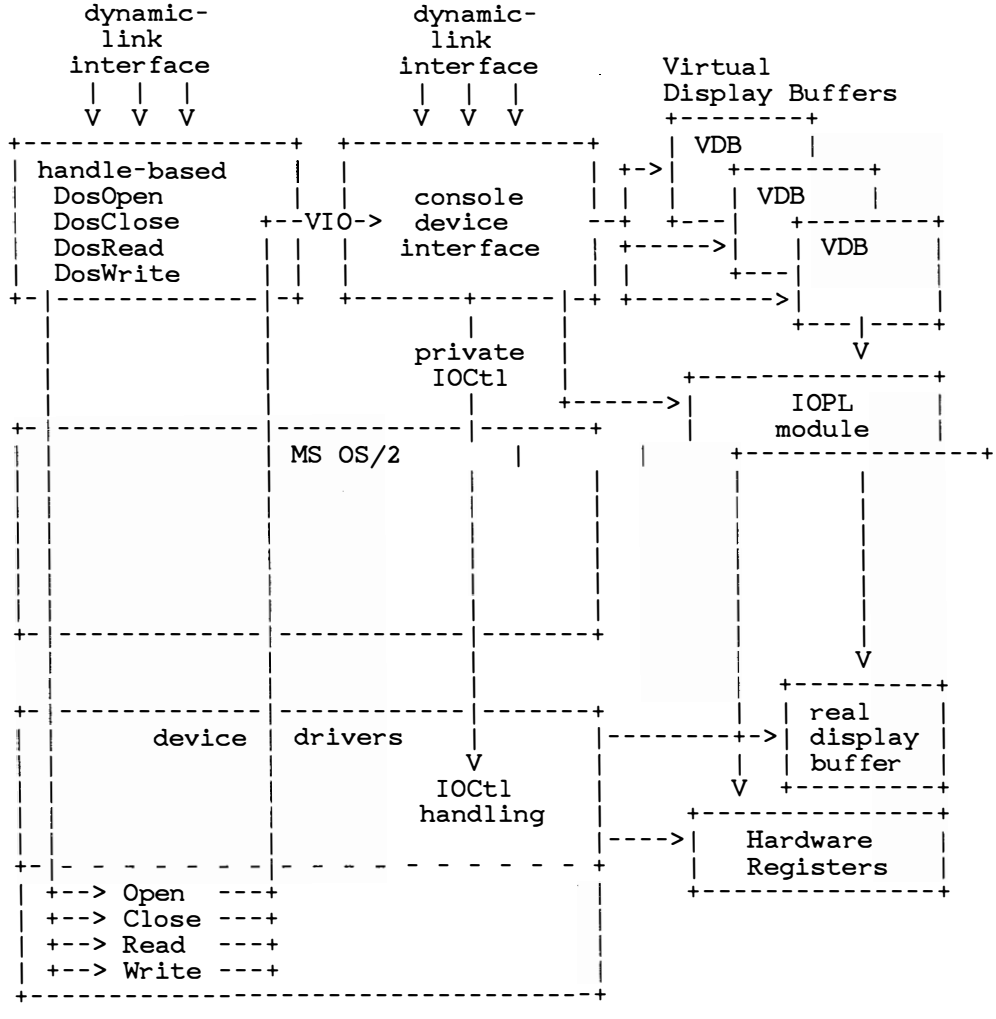


Figure 1.7 Both Handle and CDI I/O

The device driver returns handle-based I/O to the DOS system calls, which convert the I/O into a **Vio***xxx* system call. The VIO router (not shown in Figure 1.7) routes the **Vio***xxx* call to the console device interface so that the I/O can be managed in both the real and the virtual display buffers.

1.11.2 Keystroke Monitor Interface

Some applications monitor all keystrokes to provide global system functions before other applications receive the keystrokes. Examples include national language support for switching the keyboard layout and for Asian language conversion. Other examples are applications that provide a desk calculator or keystroke macro expansion. Hardware enforced protection requires that the system provide interfaces for such applications, which run as processes.

Figure 1.8 depicts the keystroke monitor interface of the keyboard device. The keystroke monitors have been previously registered as monitors. A monitor may pass the keystroke on, consume it, or replace it with one or more keystrokes.

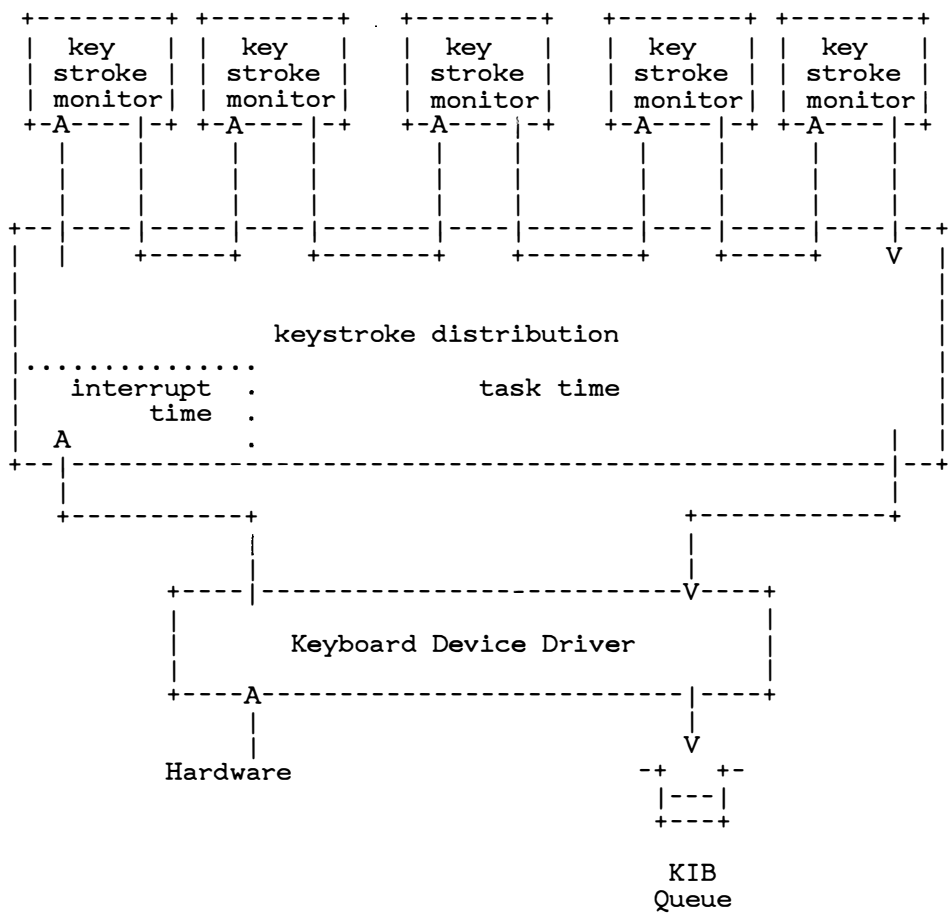


Figure 1.8 Keystroke Monitor Interface

A keyboard monitor executes at process-time and can use MS OS/2 system calls. For example, a desk calculator needs to write to the display, and a keyboard macro expander may have the macros stored in the file system.

Figure 1.9 shows the keystroke monitor interface in greater detail than is shown in Figure 1.8.

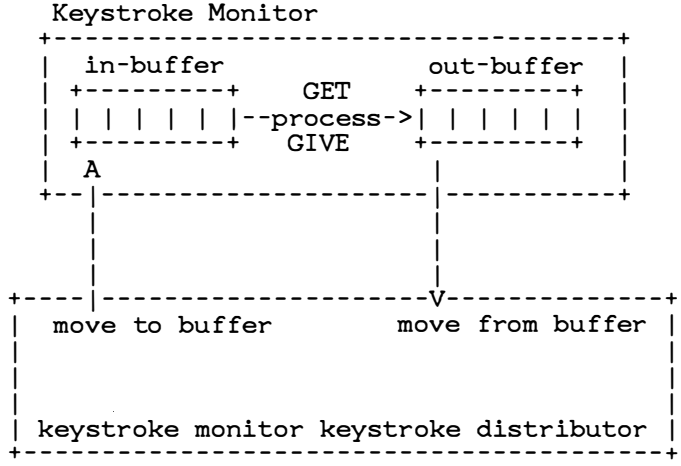


Figure 1.9 Keystroke Monitor Interface (detail)

1.12 Device Driver Initialization

Device driver initialization occurs during system initialization. During system initialization, MS OS/2 preloads its base device drivers along with the operating system.

MS OS/2 loads additional, installable device drivers by using the **device**=*command* statement in the *config.sys* file. Once the operating system has loaded the device driver, it (the device driver) is called at its strategy routine entry point with the **Init** (Command Code 0) request packet.

MS OS/2 base device drivers initialize in protected mode with each device driver strategy routine running as a single thread of execution with I/O privilege. Because of the special state of the system, a base device driver may not make dynamic-link system calls. However, the base device drivers can use the **DevHlp** services that are available at initialization time.

Installable MS OS/2 device drivers also initialize in protected mode. The device driver strategy routine for each runs under the thread of the system initialization process at application level with I/O privilege. Because of the system initialization process, the installable device drivers may make the following dynamic-link system calls at initialization time:

DosBeep
DosCaseMap
DosChgFPtr
DosClose
DosDelete
DosDevConfig
DosDevIOCtl
DosFindClose
DosFindFirst
DosFindNext
DosGetCtryInfo
DosGetDBCSEv
DosGetEnv
DosGetMessage
DosOpen
DosPutMessage
DosQCurDir
DosQCurDisk

DosQFileInfo
DosQFileMode
DosRead
DosSubSysInfoPtr
DosWrite

In addition to the set of dynamic-link system calls, certain **DevHlp** services are available to installable device drivers at initialization time. These functions are detailed in Chapter 5, "Device Helper Services."

To release code and data needed only by the device driver's initialization routine, you must follow certain guidelines. The **Init** code and **Init** data must be located at the end of the appropriate segments. Then once MS OS/2 has initialized the device, the initialization routine should set the offsets in the **Init** request packet to the end of the code segment and the end of the data segment. For more information about the **Init** request packet, see Section 3.4.1.

Note that since device driver initialization is invoked through the strategy routine, the initialization routine must not issue a **DosExit** system call, but instead must return the **Init** request packet via **DevDone**.

1.13 Compatibility with Old Device Drivers (Real-mode Session Only)

Not all old MS-DOS device drivers can be allowed to run as real-mode sessions. The supported set of old device drivers is bound by the following restrictions:

- Old block device drivers will not be permitted in the real-mode session; block device drivers must be written to the MS OS/2 interfaces.
- MS OS/2 supports only a limited set of old character device drivers. To run in the real-mode session, an old character device driver must conform to the following rules:

  1. The character device driver cannot have a hardware interrupt handler; meaning that its device must be a *polled device* rather than an interrupt-driven one (instead of depending on an interrupt from the device, the device driver must "poll," or check, the device to see if it needs to be serviced).
  2. The character device driver will be called by MS OS/2 in the real-mode session with all of the packets supported by character devices. These packets are as follows:

Code	Command
0	Init
3	IOCtl Input
4	Input (Read)
5	Non-Destructive Input No Wait
6	Input Status
7	Input Flush
8	Output
9	Output with Verify
10	Output Status
11	Output Flush

12 IOCtl Output (Write)

13 Device Open

14 Device Close

16 Generic IOCtl

These packets are subject to the same requirements as they were in MS-DOS 3.2. For example, the IOCtl bit must be set (to 1) in order to receive IOCtl requests.

The side effect of running an old character device driver is that its device may only be used in the real-mode session. Only applications running in the real-mode session may perform I/O to this device; protected-mode applications will not be able to access the device.

Certain devices cannot be exclusive to the real-mode session. Character devices in this category include mouse and clock devices. MS OS/2 does not support these old device drivers.

1.13.1 Initializing Old Device Drivers

The old character device drivers are installed in the same manner as they were under MS-DOS. The device driver program file is specified in the configuration command, **device**=*command*.

MS OS/2 loads and initializes old device drivers in real mode. The rules for replacing old character device drivers are the same as they were in previous versions of MS-DOS: the replacement is guided by the name and attributes of the device driver. The functions that can be performed at initialization are more restrictive than for MS-DOS 3.2 (or earlier version of MS-DOS). No INT 21H functions may be performed from the device driver initialization code.

The chain of old device device drivers is maintained and kept separate from the chain of MS OS/2 device drivers.

Chapter 2
Device Driver Descriptions

2.1 Introduction

This chapter contains general descriptions and information on the operation of each major type of device driver. It is divided into two parts, one on block device drivers and one on character device drivers.

Some of the drivers described in this chapter are part of the MS OS/2 base device driver set. Others are installable through the **device=** command line in the *config.sys* file.

2.2 Block Device Drivers

The MS OS/2 block device drivers are floppy and hard disk device drivers that run in a multitasking, dual-mode environment.

MS OS/2 provides the following functions:

- Media check
- Build BPB (BIOS Parameter Block)
- Read/write requests, which may be made in either of two modes:

  Absolute mode allows the user to specify a logical sector to be used for the starting I/O location.

  The other mode requires the track and sector number to be specified. Read and write requests may also be "long" (512 + 4 bytes ECC [error correction code]).
- Write with verify
- Block removable
- Generic IOCtl, which includes the following subfunctions:

  Get/set device parameters

  Read/write/verify a track

  Format and verify a track

- Get/set logical drive map

Only the Generic IOCtl function is directly accessible to the user. The remaining functions are used by the kernel to perform its operations.

The device driver operates in user mode when it is called for initialization. In addition, the disk device driver has some special support code for real-mode applications that may issue INT 13H, INT 25H, and INT 26H functions, which also run in user mode.

2.3 Character Device Drivers

2.3.1 Console Device Drivers (Screen and Keyboard) Protected-Mode-Only

In MS OS/2, the MS-DOS generic console device driver (CON) has been replaced by two independent device drivers: *Screen* (*SCREEN$*) and *Keyboard Input* (*KBD$*). Both the *SCREEN$* and *KBD$* drivers support the MS OS/2 interrupt-driven architecture.

The *SCREEN$* and *KBD$* device drivers can be installed or they can be part of the base device driver set. In either case, they must conform to the new MS OS/2 model.

MS OS/2 recognizes the *SCREEN$* and *KBD$* drivers by the *DevStdin* and *DevStdout* attribute bits in the device header. Commands sent directly to the *SCREEN$* or *KBD$* device are routed only to that device. Reads from the *SCREEN$* device are in error, since the *SCREEN$* device cannot be opened for read.

2.3.1.1 Keystroke Monitors

Some applications need to view the raw keystrokes as they arrive from the keyboard at interrupt time. These applications may wish to consume some of those keystrokes, or they may wish to replace some keystrokes with one or more keystrokes. These options are made possible by the "keystroke monitor" function.

The keyboard device driver supports device monitors. It passes its information to the monitors in packets containing the information shown in Figure 2.1:

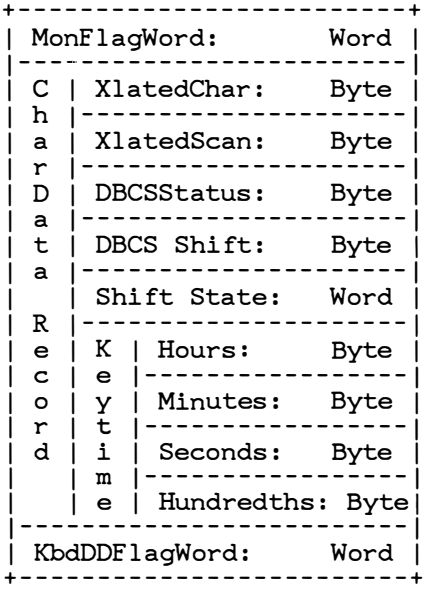


Figure 2.1 Definition of a Keystroke Monitor Data Packet

The information in the data packet is described as follows:

MonFlagWord: The low byte of the *MonFlagWord* contains the original scan code as read from the hardware.

Value	Meaning
0	This packet was inserted for other reasons; see the *KbdDDFlagWord*. Monitors pass this field untouched, but should put a zero here if they insert a packet.

The high byte of the *MonFlagWord* contains the monitor dispatcher flags with the following bitmap:

Bit	Meaning
0	Open (Not used by keystroke monitors)
1	Close (Not used by keystroke monitors)
2	Flush This is a flush packet. No other information in the packet has meaning. The monitor should flush its internal buffers and pass the packet quickly.
3–7	Reserved = 0 Should be passed untouched for packets being passed on. Should be set to zero for packets that are being inserted by a monitor.

CharData Record (as defined for **KbdCharIn**) is a ten-byte structure defined as follows:

Length	Description
BYTE	*XlatedChar* Allocated byte for an ASCII character code
BYTE	*XlatedScan* Scan code
BYTE	*DBCSStatus* National Language Support status
BYTE	*DBCSShift* National Language Support shift
WORD	*ShiftState* Shift state
4 BYTES	*Keytime* The *Keytime* field has the following four parameters:

Length	Description
BYTE	*Hours*
BYTE	*Minutes*
BYTE	*Seconds*
BYTE	*Hundredths*

KbdDDFlagWord is a Word that defines the keyboard device driver flags. It is described as follows:

Bit	Meaning
0–5	This numeric field tells the device driver that this is a key that it must act on. The number in this field, which is filled in during the translation of the scan code, allows the device driver to act on keystrokes without regard for what scan codes the keyboard uses, or what character codes the current translation process may be using. The possible values for these scan codes are detailed at the end of this list.
6	Key Break This record is generated by the release (the *break*) of the key involved.
7	Secondary The scan code prior to the one in this packet is the Secondary Key Prefix.
8	Multimake The translation process sees this scan code as a typamatic repeat of a toggle key. Since a toggle key only changes state on the first make after each key-break, no state information is changed with this key (for example, the NUMLOCK toggle bit in the shift status word is not changed, even though this may be the NUMLOCK key). If this toggle key is a valid character, it will *not* go into the KIB when this bit is set.
9	Accented This key was translated using the previous key passed, which was an accent key (see Accent Key). In the case where an accent

key is pressed and the following keystroke doesn't use the accent, a packet containing the accent character itself is passed first, with this bit set (the scan code field of *MonFlagWord* would be zero, indicating a non-key-generated record). Then a valid packet containing that keystroke is passed, without this bit set.

10–13 Reserved = 0

Monitors should pass these flags unchanged, but set them to zero in packets that they create.

14–15 Available

These bits are available for communication between monitors and are not used by the device driver in any way. It is up to the monitor applications to coordinate the use of these flags.

The following scan code values are currently defined:

- Value for keys that are always placed in the KIB:

Value	Scan code
0	No special action. Always place in the KIB.

- Values acted on prior to passing packet to monitor.

  Except for the final keystroke of the reboot and dump key sequences, all these values are passed on to the monitors and are *not* placed in the KIB. The *XlatedChar* and *XlatedScan* fields are undefined for these values.

Value	Scan code
01H	ACK This scan code was previously a keyboard acknowledgement. On IBM PC/AT attached keyboards, this value is set on a 0FAH scan code.
02H	Secondary Key Prefix This scan code was previously a prefix scan code generated by the enhanced keyboard, indicating that the next scan code is one of the secondary keys on that keyboard. Usually set on a 0E0H scan code.

03H Kbd Overrun

This scan code was previously an over-run indication from the keyboard. On IBM PC/AT attached keyboards, this value is set on a 0FFH scan code.

04H Resend

This scan code was previously a "resend" request from the keyboard. On IBM PC/AT attached keyboards, this value is set on a 0FEH scan code.

05H Reboot Key

This scan code is the key that completes the reboot key sequence. On IBM PC/AT attached keyboards, this value is used when the CONTROL-ALT-DELETE key sequence is recognized.

06H Dump Key

This scan code is the key that completes the request key sequence, Stand Alone Dump. On IBM PC/AT attached keyboards, this value is used on completion of the second consecutive press of the CONTROL-ALT-NUMLOCK key sequence *without* other keystrokes between the two presses.

07H Shift Key

This scan code, translated as a SHIFT key, affects the shift status fields of the *CharData* record but does not generate a defined character, so it is not placed in the KIB. The *XlatedChar* field is undefined.

08H Pause Key

This scan code was previously translated as the key sequence meaning "pause." On IBM PC/AT attached keyboards, this value is used when the CONTROL-NUMLOCK key sequence is recognized. The key itself is not placed in the KIB.

09H Pseudo-Pause Key

This scan code is translated into the value that is treated as the PAUSE key when the device driver is in cooked mode. On most keyboards, this is when the CONTROL-S key sequence is recognized. The key itself is not placed in the KIB.

0AH	Wake-Up Key This scan code is the key, following a PAUSE or pseudo-PAUSE key, that causes the pause state to be ended. The key itself is not placed in the KIB.
0B–0FH	Reserved = 0 (Treated as undefined, see entry 3FH.)

- Values acted on after receiving a packet from the monitors.

Except where noted, these values will be placed in the KIB when the device driver is in raw mode, but not when it is in cooked mode. Also included in the following list are those values that never get placed in the KIB.

Value	**Scan code**
10H	Accent Key This scan code was translated as a key to be used in translating the *next* key to come in. The packet containing this value is passed when the accent key is pressed, but is not put into the KIB (unless the accented bit is on). This is done in order to see what the next key is before performing an action. If the next key is one that can be accented with this key, it will be passed by itself, with the accented bit on. If that next key cannot be accented with this key, two packets are passed. The first contains the character to print for the accent itself, this value (Accent key), and the accented flag (which says it is okay to put it in the KIB). The second packet contains a regular translation of that following key.
11H	Break Key This scan code is translated as the key sequence meaning break. On IBM PC/AT attached keyboards, this value is used when the CONTROL-BREAK key sequence is recognized.
12H	Pseudo-Break Key This scan code is translated into the value that is treated as the BREAK key when the device driver is in cooked mode. On most keyboards, this is when the CONTROL-C key sequence is recognized. Note that the event generated by pressing this key is separate from that generated by pressing the BREAK key.

13H Print Screen Key

This scan code is translated as the key sequence meaning PRINTSCREEN. On IBM PC/AT attached keyboards, this value is used when the SHIFT-PRINTSCREEN key sequence is recognized.

14H Print Echo Key

This scan code is translated as the key sequence meaning "print echo." On IBM PC/AT attached keyboards, this value is used when the CONTROL-PRINTSCREEN key sequence is recognized.

15H Pseudo-Print Echo Key

This scan code is translated into the value that is treated as the print echo key when the device driver is in cooked mode. On most keyboards this is when the CONTROL-P key sequence is recognized.

16–2FH Reserved = 0

(treated as Undefined, see entry 3FH)

- Values for packets not generated by a keystroke:

Value **Scan code**

30H Status Change

This packet is generated when an IOCtl call to the device driver changes one of the states (shift status, DBCS, etc.) included in *CharData* records. The scan code field of the *MonFlagWord* of this packet is zero, and the *XlatedChar* and *XlatedScan* fields of the *CharData* record are undefined.

31H Written Key

This packet is generated when a **DosWrite** to *KBD$* passes this character to the device driver. The scan code field of *MonFlagWord* is zero, as is the *XlatedScan* field in the *CharData* record (unless the written character was previously a two-byte extended character, such as a function key value). The shift status fields in the *CharData* record are the current states at the time the write was processed by the device driver.

Note

If this value is set, the character contained in such a packet is not examined upon return from the monitors. The record will always be written to the KIB.

- Reserved = 0

(Treated as undefined, see entry 3FH)

- Value for keys that the translation process does not recognize:

Value	Scan code
3FH	Undefined This scan code, or its combination with the current shift state, is not recognized in the translation process.

When the **DosMonReg** call is used with keyboard devices, the index indicates the screen group in the range $0 \leq index \leq MaxScreenGroup$. Zero indicates the screen group of the calling thread. For more information about these settings, see the **DosMonReg** call in the *Microsoft Operating System/2 Programmer's Reference.*

2.3.1.2 Special Key Processing

By examining each incoming keyboard character, MS OS/2 determines whether the character will cause an asynchronous signal to be sent to some process (for example, CONTROL-C). The keyboard device driver responds to the following key sequences:

Key Sequence	Mode	Event signaled
CONTROL-C	Binary	Nothing
	ASCII	Event CONTROL-C
CONTROL-BREAK	Binary	Event CONTROL-BREAK
CONTROL-SCROLL LOCK	ASCII	Event CONTROL-C
CONTROL-S	Binary	Nothing
	ASCII	Event CONTROL-SCROLL LOCK

CONTROL-SCROLL LOCK	Binary	Event CONTROL-SCROLL LOCK
	ASCII	Event CONTROL-SCROLL LOCK
CONTROL-P	Binary	Nothing
CONTROL-PRINTSCREEN	ASCII	Event CONTROL-PRINTSCREEN
CONTROL-ALT-DELETE		System initialization
CONTROL-ALT-NUMLOCK		Pressed twice means system dump
PRINT SCREEN		Print the screen

In addition to passing individual characters, the driver must specifically recognize the character (or character sequence, or other special action) that indicates the special signal, or *hot key,* to the session manager. When this event is recognized, the **Signal_Event** function is called.

Note that the system hot key is defined by the IOCtl Set Session Manager Hot Key.

2.3.1.3 Compatibility Operations

The compatibility environment runs in its own screen group, and the screen group mechanism keeps MS OS/2 programs from affecting the compatibility screen. When the compatibility environment is no longer in the foreground, the old application is frozen, which keeps it from affecting the MS OS/2 screen. The compatibility environment has its own independent *SCREEN*$ and *KBD*$ device driver in the form of a compatibility-mode *CON* driver. This driver is compatible with the ROM INT 10H and INT 16H entries and their respective low-memory data structures.

2.3.2 The *CLOCK*$ Device

MS OS/2 assumes that the *Complementary Metallic Oxide Semiconductor* (CMOS) real-time clock is available in the system. The *CLOCK*$ device defines and performs functions like any other character device except that it is identified by a bit in the attribute word. Since MS OS/2 uses this bit to identify the device, the device may take any name. In MS OS/2, this device is named *CLOCK*$ to avoid possible conflicts with any files named *CLOCK*.

MS OS/2 on the IBM Personal Computer AT makes use of the clock/calendar chip for its clock ticks. This device is not available on other models of the PC line and therefore is not programmed by MS-DOS 3.*x* applications. When the foreground process is running in real mode, the regular 18.2 Hz clock ticks arrive and are intercepted and/or disposed of in a manner compatible with MS-DOS 3.*x*. When the real-mode process is running in the background, the 18.2 Hz clock is masked off. The clock/calendar clock continues to run in real and protected modes.

The *CLOCK*$ device itself—the driver that sets and returns the date and time of day— is dual-mode and services both modes. The *CLOCK*$ device cannot be reserved; the date and time can be set from either mode.

The *CLOCK*$ device is unique in that MS OS/2 reads or writes a six-byte sequence that encodes the date and time. Writing to this device sets the date and time, and reading from it gets the date and time.

Figure 2.2 illustrates the binary time and date format used by the *CLOCK*$ device:

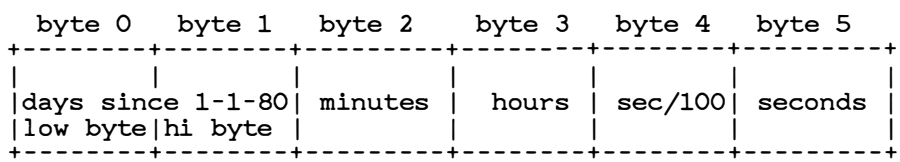


Figure 2.2 Binary Time and Date Format

CLOCK$ Device Time and Date Fields

The *CLOCK*$ device driver sets and maintains the following fields in the GDT information segment:

Time:

Time from 1-1-1970, in seconds

Milliseconds

Hours

Seconds

Hundredths

Timer interval

Date:

Day

Month

Year

Day-of-week

The *CLOCK*$ device driver ensures that the date, time from 1–1–70 in seconds, and time of day (hours, minutes, seconds) fields are synchronous with the CMOS clock, and that the hundredths of seconds field is synchronous with the seconds field. The milliseconds field is an overflowing, free-running milliseconds counter.

2.3.3 Printer Device Drivers

The printer device driver runs in a multitasking, bimodal environment.

MS OS/2 provides the following functions:

- Print character, which returns the following information if an error occurs:

  Out of paper
  Device not ready
  I/O error
  Interrupted while doing output

- Get/set frame control
- Get/set infinite retry
- Initialize printer

- Get printer status, which returns the following information:

  Busy/not busy
  Acknowledge
  Out of paper
  Selected
  I/O error
  Timeout

The printer device driver supports device monitors, and passes its information to the monitors in packets which contain the following information:

```
+-----------------------------+
| WORD      Flags             |
|-----------------------------|
| WORD      PID               |
|-----------------------------|
| n BYTES   Data bytes        |
+-----------------------------+
```

where:

Flags shows whether the operation is one of open, close, or flush for the device monitor.

PID is the process identification number of the operation.

Data bytes is an *n*-byte structure containing the data to be passed in the monitor information packet.

When called for initialization, the device driver operates in user mode. In addition, the printer device driver has some special support code for real-mode (3.*x*) applications that can issue INT 17H functions. These also run in user mode.

2.4 Asynchronous Communications Device Driver

The primary function of an asynchronous device driver is to be able to read and write data to and from one or more communications ports, while minimizing the load on the available resources. The *COM* ports are fully interrupt driven.

2.4.1 Resource Management

The installable driver performs basic exclusive access control for all processes that try to use the communications device. A device is allocated to a process when that device exists and is available, and when one of the following events occurs:

- One of the *COM1* or *COM2* devices is opened.
- Read, Write, or IOCtl is performed using *AUX* in the compatibility environment.

When a driver allocates a device, two events occur. First, the hardware interrupt vector is intercepted, and second, the corresponding presence indicator in the real-mode session at 0:400H or 0:402H is cleared in an attempt to make other programs in the real-mode session think the device does not exist.

A device is deallocated when a process closes a device that it "owns." Deallocation involves waiting for the output queue to empty (blocking the process), releasing the hardware interrupt vector, and resetting the presence indicator. If a process that does not own any devices attempts to close the *AUX* device, MS OS/2 ignores the close. This scheme should completely handle device allocation, even for processes that terminate abnormally, because MS OS/2 closes every open file upon process termination.

When a device is allocated and deallocated, the **DevHlp** routines for interrupt vector support are used to intercept and release the hardware interrupt vector. Should an attempt to intercept a vector fail, the driver assumes another process has already intercepted the vector and allocated the device for itself.

2.4.2 Read/Write Data

Unread input and waiting output are queued; when the output queue becomes full, a process writing additional data, blocks until output queue space is unavailable.

Since data is lost when the input queue has no space, programs should use **DosDevIOCtl** to specify an appropriate input queue size. Similarly, output queue size should be adjusted for an application's needs. The input queue's default length is 150 bytes; the output queue's default length is 64.

Non-blocking reads are supported by an IOCtl call that returns the number of characters available; a read call will not block if it requests no more data than is available.

2.4.3 Compatibility Mode Considerations

There are two issues for *COM* reservation in a bimodal environment: detecting that an old application is using the device so that MS OS/2 can prevent a MS OS/2 application from claiming it; and when a MS OS/2 application uses the device, preventing an old application from using it.

MS OS/2 detects an old application's use of a *COM* device by its setting of the interrupt vectors for that device. MS OS/2 checks these vectors at every context switch. Should they be changed by the old application, MS OS/2 considers the device claimed. When the vectors are found to be restored, the device is considered freed. MS OS/2 protected-mode applications cannot reserve an already claimed device.

When a MS OS/2 protected-mode application claims a *COM* device, MS OS/2 clears the associated I/O port pointer in low memory (at 40:0 and 40:2). Old applications use these values to determine the existence of the device. When the MS OS/2 program surrenders the device, the I/O port pointers are restored.

2.4.4 Installation

The asynchronous device driver is installed with a standard device driver entry in the *config.sys* file:

device=[*drive:*][*path*]*filename* [*arguments*]

2.4.5 Device Configuration

The device driver can set baud rates, parity, number of stop bits, and character size with IOCtl function calls. This information can be used by utilities such as **mode** to change the current configuration. These function calls are described in Chapter 4, "Generic IOCtl Commands."

2.4.6 Register Definitions

This section defines the content of the device registers.

Line control register

```
LC_BITS5  = 00H     ;5 data bits
LC_BITS6  = 01H     ;6 data bits
LC_BITS7  = 02H     ;7 data bits
LC_BITS8  = 03H     ;8 data bits
LC_BMASK  = 03H     ;Data bits mask
LC_STOP1  = 00H     ;1 stop bit
LC_STOP2  = 04H     ;2 stop bits
LC_SMASK  = 04H     ;Stop bits mask
LC_PNONE  = 00H     ;No parity
LC_PODD   = 08H     ;Odd parity
LC_PEVEN  = 18H     ;Even parity
LC_PSTUCK1= 28H     ;Parity bit always set
LC_PSTUCK0= 38H     ;Parity bit always clear
LC_PMASK  = 38H     ;Parity mask
LC_BREAK  = 40H     ;Transmit break
LC_RAW    = LC_PNONE + LC_STOP1 + LC_BITS8
LC_COOKED = LC_PODD + LC_STOP1 + LC_BITS7
```

Line status register

```
LS_DR     = 01H     ;Data ready
LS_OERR   = 02H     ;Over-run error
LS_PERR   = 04H     ;Parity error
LS_FERR   = 08H     ;Framing error
LS_BI     = 10H     ;Break interrupt
LS_THRE   = 20H     ;Tx holding register empty
LS_TSRE   = 40H     ;Tx shift register empty
```

Modem control register

```
MC_DTR    = 01H     ;Data terminal ready
MC_RTS    = 02H     ;Request to send
MC_OUT1   = 04H     ;Output 1
```

```
MC_OUT2  = 08H      ;Output 2
MC_LOOP  = 10H      ;Loopback mode
```

Modem status register

```
MS_DCTS  = 01H      ;Delta clear to send
MS_DDSR  = 02H      ;Delta data set ready
MS_TERI  = 04H      ;Trailing edge of ring
MS_DRLSD = 08H      ;Delta receiver line signal
MS_CTS   = 10H      ;Clear to send
MS_DSR   = 20H      ;Data set ready
MS_RI    = 40H      ;Ring indicator
MS_RLSD  = 80H      ;Receiver line signal detect
```

2.5 Pointing Device Drivers (Mouse Drivers)

2.5.1 Supported Mouse Devices

Mouse device drivers are supported in both real and protected mode for the following devices:

- Microsoft InPort (parallel) Mouse for IBM Personal Computers (Model 037-299)
- Microsoft Bus (parallel) Mouse for IBM Personal Computers (Model 037-099, 100ppi)
- Microsoft Bus (parallel) Mouse for IBM Personal Computers (Model 037-199, 200ppi)
- Microsoft Mouse (serial) for IBM Personal Computers (Model 039-099, 100ppi)
- Microsoft Mouse (serial) for IBM Personal Computers (Model 039-199, 200ppi)
- PC Mouse (serial) by Mouse Systems (Part number 900120-214)
- Visi-On Mouse (serial) (Part number 69910-1011)

While the MS OS/2 mouse device drivers support both real-mode and protected-mode applications, the means by which real-mode applications access the mouse differ from those of protected-mode applications.

Real-mode applications must use the Interrupt 33H (INT 33H) interface, and may not use the **Mou***xxx* mouse interface. Protected-mode applications must use the protected-mode **Mou***xxx* API, and may not use the INT 33H API.

2.5.2 Mouse Screen Resolutions

The screen resolution is determined either by system default or by the application issuing an explicit **VioSetMode** system call. The virtual resolution depends on the display mode selected. MS OS/2 supports the following display modes:

Mode	Type	Text-Graphics resolution	Virtual resolution	(X,Y) cell coordinates	Size
0	BW text	40 × 25	320 × 200	640 × 200	16 × 8
1	CO text	40 × 25	320 × 200	640 × 200	16 × 8
2	BW text	80 × 25	640 × 200	640 × 200	8 × 8
3	CO text	80 × 25	640 × 200	640 × 200	8 × 8
4	Graphics		320 × 200	640 × 200	2 × 1
5	Graphics		320 × 200	640 × 200	2 × 1
6	Graphics		640 × 200	640 × 200	1 × 1
7	Mono	80 × 25	720 × 350	640 × 200	8 × 8

2.5.3 Mouse Installation

Mouse support, which is installed at *Initial Program Load* (IPL) time, may be tailored according to the user's needs by using the *config.sys* file to define system mouse requirements. The following list describes the *config.sys* keywords available for customizing the installation of the mouse subsystem and related mouse device drivers.

- The **serial**=*keyword* statement specifies the communications port to which a serial mouse device is connected. If this *keyword* is not present (for serial mice), the default used is *COM1*. Otherwise, you must specify either *COM1* or *COM2*. This *keyword* is not valid for parallel mice (Microsoft Bus Mouse, for example).

- The **qsize**=*keyword* statement specifies the event queue length to be used for all protected-mode screen groups. If this *keyword* is not present, a default of 10 (maximum queue elements) is used. If you specify a queue length the *keyword* must be followed by a signed integer in the range:

  $1 \leq integer \leq 100$

  Each queue element occupies 10 bytes. Hence, the default event queue size allocates 100 bytes of event queue buffer space per screen group, with a maximum of 16 screen groups for mouse support at any one time. This limit holds even if the number of screen groups specified for the system is greater than 16. If an attempt is made to open mouse support in more than 16 screen groups, an error will occur.

- The **device**=*keyword* statement specifies the name of the mouse device driver. This statement uses an additional parameter, the statement **mode**=*keyword,* to allow the user to specify whether mouse support is required for real-mode-only, protected-mode-only, or both modes.

  The acceptable *keyword* values for **mode** are as follows:

  B Both real- and protected-mode support

  P Protected-mode support only

  R Real-mode support only

  The default setting for **mode** is B (Both). Consequently, if **mode** is not specified, both real- and protected-mode device driver support are loaded into the system.

2.5.4 Mouse Hot Button (Hot Key)

The mouse hot button (hot key) is normally defined or nullified by the protected-mode command shell.

Protected-mode applications may query the setting of the hot button by using the **MouGetHotKey** call.

If a hot key is defined, it is system-wide. When the hot button is pressed, the mouse device driver notifies the shell of the event. This notification takes place in both real and protected mode. If the system is currently operating with the compatibility mode in the foreground, it will switch into protected mode to display the shell panel.

2.5.5 Mouse Device Driver Packaging

Each of the supported devices is supplied with a named device driver that contains both real- and protected-mode functionality. The device drivers are:

Filename	Device
mousea00.sys	PC Mouse by Mouse Systems (Part number 900120-214)
mousea01.sys	Visi-On Mouse (Part number 69910-1011)

mousea02.sys	Microsoft Mouse (serial) for IBM Personal Computers (Models 039-099 and 039-199)
mousea03.sys	Microsoft Bus (parallel) Mouse for IBM Personal Computers (Models 037-099 and 037-199)
mousea04.sys	Microsoft InPort (parallel) Mouse for IBM Personal Computers (Model 037-299)

During initialization, the system loads the entire device driver into storage. To determine whether both modes of support are required, the mouse device driver examines the *config.sys* file for the parameter in the **mode**=*keyword* statement (if it exists), on the **device**=*MOUSEAxx.SYS* line specifying the mouse device driver. If protected-mode is not required, the system removes the storage occupied by the the protect-only portions of the mouse device driver support.

2.5.6 Mouse Pointer-Draw Implementation

Communication between both real- and protected-mode mouse device drivers and the screen pointer-draw routine is performed through a FAR call from the mouse driver to the entry point of the screen pointer-draw routine.

Prior to issuing the call to the pointer-draw routine, the mouse device driver must set up the following conditions:

- Set DS:SI to point to the data control block of the screen group. The data control block is defined later in this section.
- Set the *screen_func* field to the desired function code.
- Issue a far call to the screen pointer-draw routine. The address to be called must be stored in the *screen_entp* field of the screen group's data control block.

When the mouse device driver calls the screen pointer-draw routine, the draw routine must issue a CLI (Clear Interrupt) instruction to disable interrupts. The mouse device driver will have disabled the IOCtls and the mouse device interrupts. After the pointer-draw routine returns to it, the mouse device driver will enable these interrupts.

All addresses to data obtained through a **DevHlp_AllocPhys** call are stored in the control blocks in 32-bit physical address form. The pointer-draw routine must convert these addresses to virtual format (selector:offset for protected mode, and segment:offset for real mode) with

the system call **DevHlp_PhysToVirt**. The resulting selector is temporary and should be used for no more than 400 microseconds. If an interrupt occurs during the 400-microsecond span, the temporary selector becomes invalid. Consequently, the screen pointer-draw routine *must*

- Disable interrupts using the CLI instruction
- Complete all operation within the 400-microsecond time limit
- Re-enable interrupts

To bypass the 400-microsecond time limit, it is possible for the screen pointer-draw routine to repetitively perform the following operations:

- Enable interrupts
- Disable interrupts
- Call **DevHlp_PhysToVirt**
- Execute draw functions
- Call *DevHlp_PhysToVirt*

However, interrupt-time operations should be as limited in scope and duration as possible to reduce the effect on the remainder of the system. Therefore, all pointer-draw operations should complete as quickly as possible, without exceeding a single 400-microsecond interval.

The screen pointer-draw routine supports the following *screen_func* functions:

Value	Routine
0	**DrawPointer** (Bimodal)
1	**RemovePointer** (Bimodal)
2	**FreePointerMemory** (Bimodal)
3	**CheckModeProtect** (Protected-mode-only)
4	**CheckModeReal** (Real-mode-only)
5	**GetPointerMemory** (Bimodal)

The following table outlines the interrupt state of the 8259 interrupt controller for the commands that may be executed via calls from the mouse device driver to the screen pointer-draw routine. The table also describes the execution modes from which the commands may be called.

Function type	Execution modes
DrawPointer	Interrupt, User, Kernel
RemovePointer	Interrupt, User, Kernel
FreePointerMemory	User, Kernel
CheckModeProtect	Kernel
CheckModeReal	User
GetPointerMemory	User, Kernel

The function types are described as follows:

DrawPointer draws the current mouse pointer image (cursor), provided that it is not within the area defined by the collision area's definition fields.

RemovePointer removes the current mouse pointer image (if it was visible) from the screen.

FreePointerMemory is issued only after a **RemovePointer** to free both the current pointer-image buffer and its associated screen-restore buffer.

CheckModeProtect verifies the requested *mode_data* structure values as either supportable or unsupportable:

- If the requested protected-mode screen mode is unsupportable, the mouse screen device driver sets an error code of 1 in the AX register, and then returns.
- If the requested mode is supportable, the function sets the control block mode data fields accordingly, and sets a valid return code of 0 in AX.

  Mode_data is a 12-byte data structure pointed to by the ES:DI address. The *mode_data* structure is defined in Section 2.4.7, "Mouse Device Driver Control Blocks."

CheckModeReal verifies the requested *mode_data* structures values as either supportable or unsupportable:

- If the requested real-mode screen mode is unsupportable, the mouse screen device driver sets an error code of 1 in the AX register, and then returns.

- If the requested mode is supportable, the mouse screen device driver sets the control block *mode_ data* fields, and sets a valid return code of 0 in the AX register.

  Mode_ data is a one-byte data structure pointed to by the ES:DI address. The *mode_ data* for real mode is the one-byte *Len* field of the *mode_ data* structure defined in Section 2.4.7, "Mouse Device Driver Control Blocks."

GetPointerMemory is issued only after a **FreePointerMemory** call so that the mouse screen device driver can allocate memory for both the new pointer-image buffer and its associated screen-restore buffer.

This function receives the address of the pointer definition control block in the ES:DI address. The pointer definition control block contains two addresses:

- The address of the pointer-image buffer
- The address of the pointer definition record

If the pointer-image data is incomplete or unsupportable, or if this function can't get the memory required to copy the pointer-image buffer, an error code of 1 is returned in the AX register.

If the image data is correct, this function

- Copies the pointer definition record data into the control block
- Copies the pointer-image buffer
- Sets a valid return code of 0 in the AX register

2.5.6.1 Restrictions on Graphics Pointer Images

To maximize pointer-image draw performance, there are restrictions on defining graphics pointer images. These limitations are as follows:

- Graphics modes that use 320× 200 resolution require that pointer images be defined with four pixels per byte.
- Graphics modes that use 640× 200 resolution require that pointer images be defined with eight pixels per byte.

In other words, graphics pointer images must be defined in byte-width multiples. The pointer-draw routine will accept non-byte width definitions, but these definitions may cause peculiar pointer images to appear on the screen.

The pointer image does not have to be drawn on the screen by the pointer-draw routine. The pointer-draw routine can dispatch a process at level 2 or 3, and have that process take over pointer-draw operations.

This approach may be of particular interest to those subsystems and environment managers that conduct a large number of screen or processing functions for each interrupt.

2.5.6.2 Mouse Device Driver Default Pointers

The default pointer images supplied by the system are the same for real- and protected-mode mouse pointer images. MS OS/2 supplies two images:

- Default text image
- Default graphics image

The *default text image* is defined as a one word reverse video block in which the screen character remains visible.

The *default graphics image* is defined an upward-pointing arrow leaning toward the left side of the screen.

The same graphics pointer image is used for both medium- (320×200) and high-resolution (640×200) graphics modes. Medium-resolution pointers may contain up to four colors. High-resolution pointers are limited to two colors, black and white.

The bit definitions of the default pointer images are as follows:

```
DefText     Struct  ;default text pointer
ANDmsk      DW      0FFFFH
XORmsk      DW      07700H
DefText     Ends

DefGrph     Struct  ;default graphics pointer
ANDmask     DB      00111111B,11111111B
            DB      00011111B,11111111B
            DB      00001111B,11111111B
            DB      00000111B,11111111B
            DB      00000011B,11111111B
            DB      00000001B,11111111B
            DB      00000000B,11111111B
            DB      00000000B,01111111B
            DB      00000000B,00111111B
            DB      00000000B,00011111B
            DB      00000001B,11111111B
```

```
                DB      00010000B,11111111B
                DB      00110000B,11111111B
                DB      11111000B,01111111B
                DB      11111000B,01111111B
                DB      11111100B,00111111B

XORmask         DB      00000000B,00000000B
                DB      01000000B,00000000B
                DB      01100000B,00000000B
                DB      01110000B,00000000B
                DB      01111000B,00000000B
                DB      01111100B,00000000B
                DB      01111110B,00000000B
                DB      01111111B,00000000B
                DB      01111111B,10000000B
                DB      01111111B,11000000B
                DB      01111100B,00000000B
                DB      01000110B,00000000B
                DB      00000110B,00000000B
                DB      00000011B,00000000B
                DB      00000011B,00000000B
                DB      00000001B,10000000B
DefGrph         End
```

2.5.7 Mouse Device Driver Control Blocks

Internal mouse pointer control blocks are described in the following structures. These structures are used by both the real- and protected-mode mouse device drivers and by the screen pointer-draw routines.

```
Key to Notes
------------

M   = Mouse device driver access only
P   = Pointer-draw device driver access only
BM  = Both peek - only mouse device driver may modify
BP  = Both peek - only pointer device driver may modify
MON = Mouse device driver + monitor

-------------------------------------------------------------

;
; Mouse Screen Group Data Area Template (112-byte structure)
;
;
; This control block is passed during calls from the mouse
; device driver to the pointer-draw device driver.
;
; DS : SI points to this control block
; -------

scrgp_data   STRUC
;
```

```
;   Screen group control data sub-table  (next 36 bytes)
;
  Rowscale_FactDW ?  ; M   Row coordinate scale factor
  Colscale_FactDW ?  ; M   Column coordinate scale factor
  Row_Remain    DW ?  ; M   Row coordinate move remainder
  Col_Remain    DW ?  ; M   Column coordinate move remainder
  D_Status      DW ?  ; M   Device status flags
  E_Mask        DW ?  ; M   Enabled event table
  Hdle_Cntr     DW ?  ; M   Number of active device handles
  E_Queue       DW ?  ; M   Event queue DS starting offset
  Eq_Head       DW ?  ; M   Event queue head displacement
  Eq_Tail       DW ?  ; M   Event queue tail displacement
  Eq_Size       DB ?  ; M   Number of used elements in queue
  Chain_Size    DB ?  ; M   Number of monitors in chain
  Chain_Hdle    DW ?  ; M   Monitor chain handle
  Screen_Entp   DD ?  ; M   Screen driver entry point address

  Screen_Tble   DD ?  ; P   @ to screen drivers data table
  Screen_Func   DW ?  ; B   Screen driver function code
  Screen_DS     DD ?  ; BM  Screen driver data segment address
                      ;     Stored as offset/selector or
                      ;     Stored as offset/segment
;
;   Monitor chain output buffer (next 12 bytes)
;
  MFlags        DW ?  ; MON monitor flags
  EMask         DW ?  ; MON event occurrence mask value
  Thrs          DB ?  ; MON event time stamp (hours)
  Tmin          DB ?  ; MON event time stamp (minutes)
  Tsec          DB ?  ; MON event time stamp (seconds)
  Thsec         DB ?  ; MON event time stamp (hundredths)
  Row_Pos       DW ?  ; MON current pointer row coordinate
  Col_Pos       DW ?  ; MON current pointer column coordinate
;
;   Display info data sub-table (next 52 bytes)
;                     Display Mode fields
;
  Length        DW ?  ; BP  Length of display mode fields (bytes)
  Mtype         DB ?  ; BP  Mono text/color text/color graphics
  Color         DB ?  ; BP  Number of color bits (graphics type only)
  TCol_Res      DW ?  ; BP  Column resolution (text)
  TRow_Res      DW ?  ; BP  Row resolution (text)
  GCol_Res      DW ?  ; BP  Column resolution (graphics)
  GRow_Res      DW ?  ; BP  Row resolution (graphics)
  Col_Cell_SizeDW ?  ; BP  Graphics column resolution/ text column resolution
  Row_Cell_SizeDW ?  ; BP  Graphics row resolution/ text row resolution
;
;    Mouse Pointer fields
;
  Ptr_Flags     DW ?  ; P   Pointer-image visible/hidden
  Ptr_Height    DW ?  ; P   Height of pointer image (resolution units)
  Ptr_Width     DW ?  ; P   Width of pointer image (resolution units)

  Ptr_Row_Pos   DW ?  ; BM  Current pointer row coordinate
  Ptr_Col_Pos   DW ?  ; BM  Current pointer column coordinate

  Ptr_Row_Ref   DW ?  ; P   Pointer-shape reference pixel row coordinate
  Ptr_Col_Ref   DW ?  ; P   Pointer-shape reference pixel column coordinate
  Ptr_Image_BufDD ?  ; P   Physical address to pointer-image buffer
  Ptr_Buf_Len   DW ?  ; P   Pointer-image buffer length (bytes)
  Ptr_Imagelen  DW ?  ; P   Pointer-image length (bytes)
  Ptr_Imageoff  DW ?  ; P   Pointer-image offset (bytes)
  Ptr_Linelen   DW ?  ; P   Pointer-image line length
```

```
  Ptr_Skiplen   DW ?  ; P   Pointer-image skip length
  Tot_Linelen   DW ?  ; P   Pointer total line length
  Ptr_Savstart  DW ?  ; P   Pointer save start pointer
  Ptr_Savend    DW ?  ; P   Pointer save end pointer
  Ptr_Savstrtodd  DW ?  ; P   Pointer save start odd pointer
  Ptr_SavendoddDW ?  ; P   Pointer save end odd pointer
;
;   Collision Area fields
;
  Area_Flags    DW ?  ; BM  Area flags (area defined/undefined)
  Area_Row_Pos  DW ?  ; BM  Area starting row coordinate position
  Area_Col_Pos  DW ?  ; BM  Area starting column coordinate position
  Area_Row_End  DW ?  ; BM  Area ending row coordinate position
  Area_Col_End  DW ?  ; BM  Area ending column coordinate position
scrgp_data    ENDS

---------------------------------------------------------------

;                   Pointer Definition Record Template
;                        (12-byte structure)
;
; Following is used to pass data about the pointer image
; during the MouSetPtrShape call.
;
; Ptr_def_cb structure points to this control block.
; ----------

ptr_template STRUC
   buf_len      DD ?  ; BM  Pointer-shape buffer byte length
   width        DW ?  ; BM  Pointer-shape width dimension
   height       DW ?  ; BM  Pointer-shape height dimension
   col_hot      DW ?  ; BM  Pointer hot spot pixel column coordinate
   row_hot      DW ?  ; BM  Pointer hot spot pixel row coordinate
ptr_template ENDS

---------------------------------------------------------------

;                   Mode Data Record Template
;                (12-byte structure for protected mode)
;
; Following is used to pass setmode data only. After setmode
; a copy of this is in the equivalent fields in the first
; control block.
;
; ES : DI points to this control block
; -------

Mode_Data     STRUC
   len          DW ?  ; BM  Length of this data structure
   m_type       DB ?  ; BM  Display Mode type value
   m_color      DB ?  ; BM  Number of color bits
   tcol_res     DW ?  ; BM  Text column resolution
   trow_res     DW ?  ; BM  Text row resolution
   gcol_res     DW ?  ; BM  Graphics column resolution
   grow_res     DW ?  ; BM  Graphics row resolution
Mode_Data     ENDS

---------------------------------------------------------------

;                     GetPointerMemory Data Structure
;
; Used to pass pointer image and associated control block from mouse
; device driver to pointer-draw device driver on MouSetPtrShape only.
```

```
;
; ES : DI points to this control block
; -------

ptr_def_cb  STRUC
     addr1      DD ?  ; BM  Address to pointer definition record
     addr2      DD ?  ; BM  Address to pointer-image buffer
ptr_def_cb  ENDS
```

2.5.8 Protected-mode Mouse Support

Mouse device drivers have characteristics that are quite different from most other devices. They are read-only devices that provide *structured data* at approximately 30 events-per-second. Structured data is data that arrives as a packet containing the absolute screen position, button status (pressed or released), and other information.

The model of the protected-mode mouse driver described in this section provides a basic, machine-independent interface. Applications and environment managers may use this interface to access mouse device services.

The *Base Mouse Subsystem* (BMS) is a dynamic-link module that executes on level 3 (application level). The BMS receives all **Mou***xxx* calls (as a common entry point), and passes those calls to the appropriate handler. There may be only one handler per screen group.

Normally, the system-supplied default handler is the screen group's mouse handler. However, environment managers, and custom mouse device drivers may find it necessary to intercept **Mou***xxx* calls for various reasons. A particular custom handler may service many different screen groups, provided it is registered (using **MouRegister**) with each of those screen groups.

The mouse device driver updates the pointer image by using the functions supplied by the display device drivers. At interrupt time, the mouse driver updates the pointer, ensuring that the pointer shape moves smoothly and promptly across the screen. The **Mou***xxx* API contains three commands, which allow the application to:

- Set the pointer shape
- Reserve a collision area where the pointer must not be drawn
- Free a collision area to the pointer device

The responsibilities of the mouse driver *must* be well separated from those of a screen driver. To maintain independence between the mouse and display devices, pointer updating functions call the display device drivers rather than attempting to draw the pointer image directly. But this separation does require custom display device drivers to conform to the pointer driver interface so that the pointer image can be drawn. The separation also requires that the application synchronizes the display.

2.5.8.1 Installing the Protected-mode Pointer-Draw Routine

The mouse pointer image's update design allows the update routine to be installed with the video subsystem. Custom and OEM video subsystems allowing display modes that are not supported by the base video subsystem may update the screen pointer image at interrupt time by providing screen pointer-draw routines for execution by the pointer device driver.

The mouse pointer device driver calls the screen pointer-draw routines at interrupt time. Then at IPL time, the mouse pointer driver installs the screen pointer-draw routines as character device drivers.

The steps for installing a screen pointer-draw routine are as follows:

- Pointer-draw routines are installed at IPL time by including them on a **device**=*keyword* statement in the *config.sys* file as named character device drivers. No special mechanism is needed.

- The display (screen) IOCtl (Category 03H, Function 72H) must be supported by the named pointer-draw device driver. This IOCtl allows the mouse subsystem router/handler to query the pointer-draw device driver for the entry (FAR call) address.

- When an application uses the **MouOpen** call to open a mouse handle, the mouse subsystem handler/router will inspect the stack to determine whether the call specified a non-system pointer-draw driver name.

  If the pointer is zero, the mouse subsystem uses the default (system supplied) device driver. This means that the screen group is restricted to display modes 0 through 7.

  If the pointer is not zero, the mouse subsystem follows the pointer to get the ASCIIZ name of the pointer-draw device driver.

- The mouse subsystem handler/router opens the pointer-draw character device driver by using **DosOpen**. Then, using the device handle returned by the open call, the handler/router issues the Category 03H, Function 72H screen control IOCtl to get the entry address (selector:offset) of the pointer-draw device driver. This is the entry address to which the mouse device driver issues far calls when mouse interrupts occur.

- The mouse subsystem handler/router then performs a **DosOpen** call on the mouse device driver. The return value from this call will be a standard MS OS/2 device handle.

- The mouse subsystem handler/router uses the **DosOpen** device handle to pass the entry address of the pointer-draw routine to the mouse device driver. This is done by using the Category 07H, Function 5AH Mouse Control IOCtl addressed to the mouse device driver's handle.

- The mouse device driver calls the pointer-draw routine with each interrupt without their being linked prior to boot time. Linkage will be established (through the IOCtls) for each pointer-draw device driver specified by a **MouOpen** call.

There may be one, and only one, pointer-draw routine (driver) for each screen group.

This mechanism applies only to protected mode.

2.5.8.2 Protected-mode Mouse Driver Handler/Router

There are three main design concepts to a MS OS/2 mouse device driver:

- **Mou***xxx* API functionality, which allows applications to avoid the specifics of the low-level IOCtl interface
- Circular buffers to receive events
- Cursor management interface

The mouse driver is installed as a character device, with the name *mouseaxx.sys*. The **MouOpen** system call initializes the device (sets initial coordinates and checks for presence of a mouse).

The **Mou***xxx* interface lets the caller obtain information about the current state of the mouse and set parameters. It also allows an application to determine which events are to be passed into the device's circular buffer.

The circular I/O buffer is a high-efficiency buffer shared by the mouse device driver and all client applications in the screen group. There is only one queue per screen group, no matter how many applications within the screen group are using the mouse device. The driver uses this interface to provide "events." The caller can specify what constitutes an event; for example, button presses or mouse movement. Events are time stamped so that a higher-level interface library package can provide such features as double clicking.

2.5.8.3 Protected-mode Mouse Coordinates

The absolute position of the mouse events record is relative to the top-left corner (1,1) of the display screen. This means that the units in which the mouse position is reported depend on the display mode in which the screen group is executing. There are two possibilities:

- In text mode, the pointer position is reported in character units.
- In graphics mode, the pointer position is reported in pixels.

By supplying pointer coordinates as offsets to the absolute screen position, high-level-library support for translating data into an absolute coordinates may no longer be necessary. But for systems that need to operate in terms of relative displacements, the library support or the application may determine how far the pointer was moved since the last mouse event was read.

2.5.8.4 Protected-mode Mouse Motion

The unit of motion for a mouse is known as a "mickey," which is similar to the pixel, the unit of addressability on a screen. The mouse driver provides a call to determine the number of units of motion (mickeys) per centimeter, so that a window manager or other package can relate motion to a physical screen.

2.5.8.5 Protected-mode Mou*xxx* and IOCtl Calls

Applications need not concern themselves with the mouse IOCtl interface. Instead, the MS OS/2 mouse device driver should be accessed by applications via the **Mou***xxx* API. Only environment managers and custom mouse device drivers need be aware of the IOCtl interface. For details on the mouse IOCtl interface, see Chapter 3, "Device Driver Commands."

The Category 7 mouse device driver IOCtl functions have application-level **Mou***xxx* API equivalent functions. The following IOCtl function codes and their API equivalents are currently defined:

IOCtl	Mou API Function	Description
50H		Allow Pointer Drawing after Screen Switch
51H		Update Screen Display Mode
52H		Notify Screen Group Switch
53H	MouSetScaleFact	Set New Scaling Factor
54H	MouSetEventMask	Set New Event Mask
55H	MouSetHotKey	Set System Hot Key
56H	MouSetPtrShape	Assign New Pointer Shape
57H	MouDrawPtr	Unmark Collision Area
58H	MouRemovePtr	Mark Collision Area
59H	MouSetPtrPos	Set Pointer Screen Position
5AH		Set Protected-mode Pointer-draw Address
5BH	MouInitReal	Set Real-mode Pointer-draw Address
5CH	MouSetDevStatus	Set Mouse Status Flags
60H	MouGetNumButtons	Get Number of Mouse Buttons
61H	MouGetNumMickeys	Get Number of Mickeys/Centimeter
62H	MouGetDevStatus	Get Device Status Flags
63H	MouReadEventQue	Read Event Queue Contents
64H	MouGetNumQueEl	Get Event Queue Status
65H	MouGetEventMask	Get Current Event Mask
66H	MouGetScaleFact	Get Current Scaling Factor
67H	MouGetPtrPos	Get Pointer Screen Position
68H	MouGetPtrShape	Get Pointer-Shape Image
69H	MouGetHotKey	Get System Hot Key
N/A	MouOpen	Open Mouse Device
N/A	MouClose	Close Mouse Device
N/A	MouRegister	Install Mouse Subsystem
N/A	MouDeRegister	Deinstall Mouse Subsystem

The **MouOpen** and **MouClose** calls are mapped by the mouse subsystem router into the **DosOpen** and **DosClose** IOCtl commands, respectively.

MouRegister and **MouDeRegister** are directed to the mouse subsystem router and do not translate down to the IOCtl device driver level. For more information about the **Mou***xxx* API function calls, see the *Microsoft Operating System/2 Programmer's Reference.*

The mouse subsystem receives all **Mou***xxx* system calls. Function-specific data is passed on the user's stack in the following format:

```
+-------------------------------------------------+
| DWORD         Mouse router return address       |
|-------------------------------------------------|
| WORD          Entry-point return address        |
|-------------------------------------------------|
| WORD          Function code                     |
|-------------------------------------------------|
| DWORD         Application (caller) return address |
|-------------------------------------------------|
| n BYTES       Function-specific parameters      |
+-------------------------------------------------+
```

where:

The *Mouse router return address* is a Dword at the top of the stack.

The *Entry-point return address* is a Word specifying the entry point address to the device driver.

The *Function code* is a Word that specifies the function be performed.

The *Application return address* is a Dword that specifies the return address for the application.

The *Function-specific parameters* may range from 2 to 10 bytes long, depending on the call.

2.5.8.6 Protected-mode Mouse Events

The mouse driver provides data to the user through the standard MS OS/2 asynchronous and parallel I/O interfaces described in other sections of this manual.

Mouse events are placed in a circular I/O buffer. The conditions that generate an event can be controlled through the event mask feature described in Section 2.5.8.2, "Protected-mode Mouse Driver Handler/Router." These conditions are accessible via the **MouSetEventMask** call or its equivalent mouse IOCtl command.

Mouse events have the following format:

```
+-----------------------------------+
|  WORD    Event mask               |
|-----------------------------------|
|  DWORD   Time                     |
|-----------------------------------|
|  WORD    X absolute row           |
|-----------------------------------|
|  WORD    Y absolute column        |
+-----------------------------------+
```

where:

Event mask is a Word that indicates which events are in this record. For more information about event mask bit definitions, see the **MouGetEventMask** system call in the *Microsoft Operating System/2 Programmer's Reference.*

Time is a Dword that indicates a time stamp for the event. *Time* is provided so that higher level interface packages can provide features like double-clicking with selective timing, and so that mouse and keyboard monitor events can be synchronized. The *Time* value is in the standard MS OS/2 time-of-day format.

X/Y absolute are Words that indicate absolute motion of the pointer shape relative to the top left corner of the display screen. These coordinates may be specified in either character or pixel offsets, depending on whether the display mode for the screen group is text or graphics, respectively.

2.5.8.7 The Mouse Pointer

The mouse device driver maintains the pointer's image and location. An application must provide a pointer image that the mouse driver can use to draw the pointer for that screen group.

In addition, applications using the mouse services must ensure that the mouse device driver and the application do not attempt to update the screen at the same location, at the same time. The **Mou***xxx* API provides three calls accomplish these functions:

- **MouSetPtrShape**
- **MouDrawPtr**

- **MouRemovePtr**

The application must also synchronize pointer operations between itself and the pointer-draw routine in the mouse device driver.

For more information on the **Mou***xxx* API calls, see the *Microsoft Operating System/2 Programmer's Reference.*

2.5.8.8 The Supported Display Modes

In MS OS/2, text modes are limited to CGA-compatible modes 0, 1, 2, and 3, and monochrome mode 7. Graphics modes are limited to CGA-compatible modes 4, 5, and 6.

MS OS/2 does not support extended (enhanced) EGA modes.

2.5.9 Real-mode Mouse Support

Real-mode mouse support is based on the Microsoft INT 33H support. For MS OS/2, the number of mouse devices supported has been expanded to include the Visi-On and PC Mouse devices, in addition to the Microsoft devices.

2.5.9.1 Installing the Real-mode Pointer-draw Routine

The real-mode pointer-draw routines are installed by the **MouInitReal** call, which the command shell (system session manager) issues at IPL time. To establish addressability between the real-mode mouse device driver and the real-mode screen pointer-draw routine, **MouInitReal** takes the following steps:

- If necessary, **MouInitReal** opens the device driver named in the **MouInitReal** call. Currently, this name is not a valid parameter, so the default name of the real-mode pointer-draw routine will always be used.
- **MouInitReal** issues a screen control IOCtl, Category 03H, Function 72H, to the real/protected-mode pointer-draw device driver. This IOCtl returns the address of the pointer-draw routine's entry point.

- **MouInitReal** then issues a mouse control IOCtl, Category 07H, Function 5BH, to the real-mode mouse device driver. This IOCtl call passes the pointer-draw address obtained from the preceding screen control IOCtl, Category 03H, Function 72H, to the mouse device driver.
- **MouInitReal** then returns to the command shell with a completion code indicating the result of the real-mode mouse initialization process.

For more information about **MouInitReal**, see the *Microsoft Operating System/2 Programmer's Reference.*

2.5.9.2 Real-mode Mouse Handler/Router

The mouse handler/router is applicable only to protected-mode mouse support. In protected mode, its purpose is to direct operations among multiple screen groups. In addition, it allows the **Mou***xxx* calls to be intercepted and synchronized between multiple processes in a screen group. This interception is done by an environment manager, by the mouse subsystem, or by a sophisticated application.

Real-mode mouse support calls may be intercepted by hooking the INT 33H interrupt vector. While there are multiple protected-mode screen groups allowed, there may never be more than one real-mode screen group. Consequently, there is no need, and no support provided for, a real-mode mouse handler/router.

2.5.9.3 Real-mode Mouse Coordinates

The protected-mode mouse reports its coordinate position in absolute displacement (characters or pixels) from the upper-left corner of the screen. In a similar manner, real-mode mouse support reports mouse coordinates relative to the upper-left corner of the screen. But in contrast to protected-mode mouse support, the real-mode mouse support reports its position in virtual screen units. These units are of a constant range, regardless of the dimensions or resolution of the screen.

Row = screen height $= 1 \leq value \leq 200$
Col = screen width $= 1 \leq value \leq 640$

All coordinate positions are reported within these ranges. The relative displacement of one unit depends upon the dimensions of the screen and upon the associated resolution of the mode setting on the display while the

compatibility box is the foreground screen group.

2.5.9.4 Real-mode Mouse Motion

As with protected-mode mouse support, the unit of motion for a mouse is a "mickey," which is similar to the pixel, the unit of screen addressability.

2.5.9.5 Real-mode Mouse IOCtl Calls

The protected-mode shell (system session manager) always exists in the MS OS/2 system. The shell calls for real-mode mouse device initialization at shell initialization time. The purpose of this shell call is to determine the entry point of the pointer-draw device driver. It is this entry point that, on interrupt, the mouse device driver will call in order to update the pointer image for the real-mode screen group.

Portions of the mouse device driver are used by both the real- and protected-mode device drivers. However, one IOCtl is supported for only the real-mode mouse device driver; Category 07H, Function 5BH, Set Real-mode Pointer-draw Address. This IOCtl call is used only by the command shell, and then only at shell initialization time.

The real-mode device driver does not support any other MS OS/2 mouse IOCtl calls.

2.5.9.6 Real-mode Mouse Events

The Microsoft INT 33H real-mode mouse API detects and reports changes in the state of the mouse. The mouse support reports events such as

- Button press data
- Button release data
- Button number
- Mouse movement

2.5.9.7 Real-mode Mouse Pointer

The real-mode mouse API supports application pointer-image setting. Two commands, in particular, enable the application to modify the compatibility-box pointer image:

- Set Text Pointer Shape
- Set Graphics Pointer Shape

The text pointer shape command is used while the display is in text modes. The graphics pointer shape command is used while the display is in graphics modes.

2.5.9.8 Supported Real-mode Display Modes

In real mode, text modes are limited to CGA-compatible modes 0, 1, 2, and 3, and monochrome mode 7. Graphics modes are limited to CGA-compatible modes 4, 5, and 6.

MS OS/2 does not provide real-mode support for extended (enhanced) EGA modes.

2.6 RAM Disk Device Driver Support

MS OS/2 includes an installable RAM disk device driver. This driver, called *RDISK.SYS*, supports the following command-line configuration in *config.sys:*

device=rdisk.sys [*bbbb*] [*ssss*] [*dddd*]

where:

bbbb is the disk size in kilobytes. The default value is 64, the minimum is 16.

ssss is the sector size in bytes. The default value is 128. Allowed values are 128, 256, 512, and 1024.

dddd is the number of root directory entries. The default value is 64, the minimum is 4, and the maximum is 1024. Because one directory entry is used by the directory, the number of actual entries is one less than specified in *dddd.*

rdisk.sys is the *RDISK* device driver. It adjusts the value of *dddd* to the nearest sector size boundary. For example if you give a value of 25, and the sector size is 512 bytes, 25 will be rounded up to 32—the next multiple of 16 (there are sixteen 32-byte directory entries in 512 bytes).

Note

If there is not enough memory to create the *RDISK* volume, *RDISK* tries to make a DOS volume with 16 directory entries. This may result in a volume with a different number of directories than was specified by *dddd.*

The device driver *RDISK.SYS* calls the MS OS/2 memory manager to allocate its memory requirements.

2.7 Replaceability of Character Device Drivers

MS OS/2 character device drivers are replaceable. For base character device drivers, both the appropriate bits in the attribute field of the device driver header and the name of the character device driver must match. For example, to replace the system clock device driver, the header in the new device driver must have the clock device indicator set, as well as have the system name for the clock.

When the new device driver is loaded, MS OS/2 uses the attribute field and name to determine whether the new device driver is attempting to replace a driver already installed. If so, then the previously installed device driver is requested to "deinstall" the indicated device with **Deinstall** (Command Code 20). If the already-installed device driver refuses the **Deinstall** command, then the new device driver is not allowed to initialize. If the already-installed device driver performs the **Deinstall** command, then the new device driver is initialized. Note that the **Deinstall** command is needed for a device driver to relinquish its interrupt vectors and its allocated physical memory.

Chapter 3
Device Driver Commands

3.1 Device Driver Header

One of the ways that a device driver file differs from an ordinary *.exe* file is that the data segment of a device driver file must contain a device driver header at the top of the segment. There must be only one data segment and one code segment, and the data segment must precede the code segment so that the device driver header follows the *.exe* file header.

The device driver file image is shown in Figure 3.1

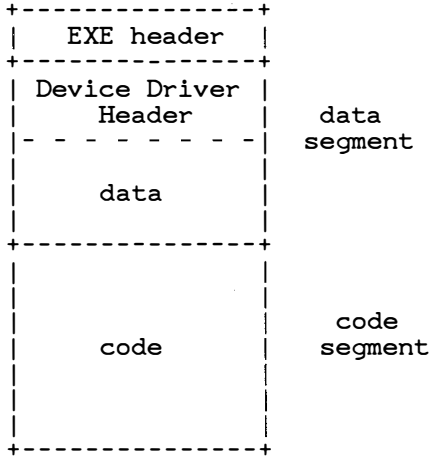


Figure 3.1 MS OS/2 Device Driver File Image

Figure 3.2 depicts the format of the device header.

DWORD	Pointer to next header
WORD	Device attribute
WORD	Offset to strategy routine
WORD	Reserved
8 BYTES	Name or Units
8 BYTES	Reserved

Figure 3.2 MS OS/2 Device Header

The *Pointer to next header* field is modified by MS OS/2 at the time the device driver is loaded. For loadable device drivers, this field should be set to –1. Note that for a character device driver that supports multiple devices, the data segment will contain a device driver header for each device. These headers must be linked together.

The *Device attribute* field describes the characteristics of the device driver and is defined in Section 3.2, "Device Attribute Word."

The *Offset to strategy routine* field is the offset of the entry point of the strategy routine. MS OS/2 uses this offset to call the strategy routine.

The *Name or Units* field contains the name of the device supported by the character device driver, or the number of units represented by a block device driver. For character devices, the name is left-justified and the remaining space is set to blanks. For block devices, the number of units can be placed in the first byte (this is optional because MS OS/2 fills it in when the driver is initialized).

Character device drivers should consider the following rule when selecting a device name: a device name takes precedence over a filename in a **DosOpen** system call. This rule means that files cannot have the same name as a character device driver, because a **DosOpen** call will always open the device instead of the file. To avoid such conflicts, name a character device driver with a character string that has an unusual last character, such as a dollar sign ($).

3.2 Device Attribute Word

Figure 3.3 defines the format of the MS OS/2 device attribute.

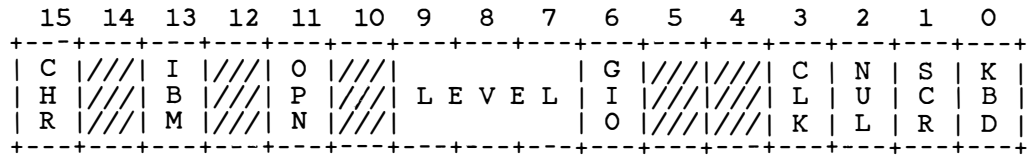


Figure 3.3 MS OS/2 Device Attribute

The bits in the device attribute word are described as follows:

Bit	Description
0	Set if this is a standard input device (Stdin)
1	Set if this is a standard output device (Stdout)
2	Set if this is the NUL device
3	Set if this is the *CLOCK*$ device
4	Reserved = 0
5	Reserved = 0
6	Set if the device accepts Generic IOCtl commands
7–9	Represents the function level, where 010 = MS OS/2 device driver
10	Reserved = 0
11	Set if the device supports removable media (block devices) or if it supports device open/close (character devices)
12	Reserved = 0
13	Set if this is a non-IBM block format (block device drivers only)
14	Reserved = 0
15	Set if this is a character device driver; zero if it is a block device driver

3.3 MS OS/2 Device Driver Request Packets

The device driver strategy routine is called with ES:BX pointing to the request packet. The interrupt routine gets the pointer to the request packet from its queue head (the device driver keeps the head of the work queue in its data segment).

The request packet consists of two parts: the 13-byte *request header* and the *n*-byte *command-specific data* field. Figure 3.4 describes the format of the request packet.

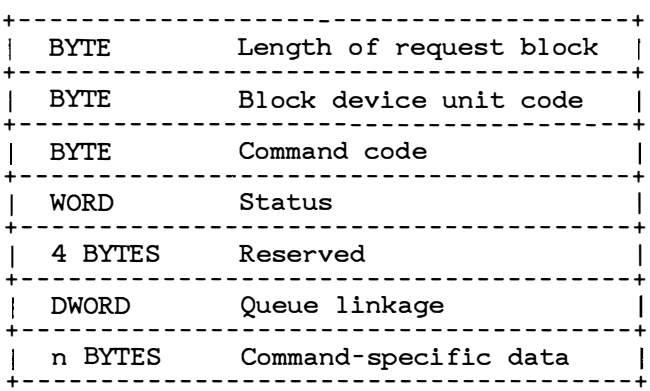


Figure 3.4 MS OS/2 Request Packet

The *Length of request block* field must be set to the total length, in bytes, of the request packet; that is, the length of the request header plus the length of the data.

The *Block device unit code* field identifies the unit for which the request is intended. This field has no meaning for character devices.

The *Command code* field indicates the requested function. The command codes are described in Section 3.4, "Microsoft Operating System/2 Device Driver Commands."

The *Status* field describes the resultant state of the request. The *Status* field is detailed as follows:

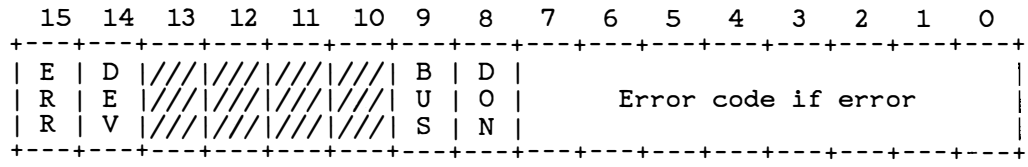


Figure 3.5 Request Packet Status Field

The bits in the *Status* field are described as follows:

Bit	Description
0–7	Error code
8	Set if Done
9	Set if Busy
10–13	Reserved = 0
14	Set if device driver defined error and bit 15 set Note: if a system-defined category, then the error returned to the caller is 0FE00H logically ANDed with the byte-length error code.
15	Set if Error

The following list details the error codes:

Code	Description
00H	Write-protect violation
01H	Unknown unit
02H	Device not ready
03H	Unknown command
04H	CRC error
05H	Bad drive request structure length
06H	Seek error

07H	Unknown media
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0DH	Change disk (logical switch)
0EH	Reserved
0FH	Reserved
10H	Uncertain media
11H	Character I/O call interrupted
12H	Monitors not supported
13H	Invalid parameter

For notes on these error codes, see the comments under "Remarks on Device Driver Errors" later in this section.

The *Reserved* field is reserved and must be zero.

The *Queue linkage* field is provided to maintain a linked list of Request Packets. The device driver may use the queue management provided by the **DevHlp** services, or it may use its own queue management. Note that since a pointer to a Request Packet is bimodal, the pointer may be used directly as the queue linkage instead of as a 32-bit physical address.

The description of the *Command-specific data* field for each device driver command is in Section 3.4, "MS OS/2 Device Driver Commands."

Remarks on Device Driver Errors

When the state of the media in the drive is uncertain, the device driver should return error code 10H. This response should *not* be returned to the **Init** command. For fixed disks, the device driver must begin in an "Uncertain media" state to have the media correctly labeled. In general, the following guidelines may be used to determine when to respond with "Uncertain media":

- When a drive-not-ready condition is detected

Note

Return "Uncertain media" (error code 10H) to all subsequent commands until a Reset Media command (Command Code 17) is received.

- When accessing removable media without change-line support, and a time delay of two or more seconds has occurred
- When the state of the change-line indicates that the media may have changed

3.4 MS OS/2 Device Driver Commands

Table 3.1 lists the code for each device driver command and whether it is used for a block or character device driver:

Table 3.1
Summary of Commands for Devices

Code	Command	Block	Character
0	Init	*	*
1	Media Check	*	
2	Build BPB	*	
3	Reserved		
4	Read (input)	*	*
5	Nondestructive Read No Wait		*
6	Input Status		*
7	Input Flush		*
8	Write (output)	*	*
9	Write with Verify	*	*
10	Output Status		*
11	Output Flush		*
12	Reserved		
13	Device Open	*	*
14	Device Close	*	*
15	Removable Media	*	
16	Generic IOCtl	*	*
17	Reset Media	*	
18	Get Logical Drive Map	*	
19	Set Logical Drive Map	*	
20	Deinstall		*
21	Port Access		*
22	Partitionable Fixed Disks	*	
23	Get Fixed Disk Map	*	
24–26	Reserved		

The following sections detail each of these device driver commands.

Init: Initialize Device (Command Code 0)

Purpose:

Command Code 0 initializes the device.

Format of Request Packet:

```
+-------------------------------------+
| 13-BYTE   Request header            |
+-------------------------------------+
| BYTE      Number of units           |
+-------------------------------------+
| DWORD     Pointer to DevHlp         |
| Offset ending code address on exit  |
| Offset ending data address on exit  |
+-------------------------------------+
| DWORD     Pointer to Init arguments |
|           set to BPB array on exit  |
+-------------------------------------+
| BYTE      Drive number              |
+-------------------------------------+
```

Remarks:

On entry, the request packet contains the following fields:

- *Pointer to DevHlp* is set to point to the **DevHlp** strategy routine entry point.

  The **DevHlp** entry point is a bimodal address that is called to invoke a service specified in the DL register.

- *Pointer to Init arguments* specifies the device specific information needed to initialize the device.

  For base device drivers (those preloaded with the operating system), the arguments are passed by the system initialization process and are device specific.

  The initialization arguments for installable device drivers are obtained from the **device=** line of the *config.sys* file. They allow the device driver to use configurable parameters to initialize the driver and the device.

- The *Drive number* for the first unit of a block device driver is set.

On completion of initialization, the device driver must set the following fields in the request packet as follows:

- Set the *Number of units* (block devices only) to the number of logical block devices or units available.

  The operating system will assign sequential drive letters to these unit. Character device drivers will set this field to zero.

- Set the first word offset (*Offset ending code address*) in the *Pointer to DevHlp* field to the end of the code segment, and the second word offset (*Offset ending data address*) to the end of the data segment.

  This field must be filled with the offsets of the code and data segments for both block and character devices. This allows a device driver to release code and data that is used only by the device driver's initialization routine. (First, the initialization code and data must be located at the end of the appropriate segments. Then, as the final step in initialization, the device driver sets the offsets to the end of the code segment and the end of the data segment.)

  This also permits a device driver to load with a maximum-sized data segment (64KB) and then allows the kernel to release the amount that it does not need.

- *Pointer to BPB array* (block devices only) is set for the logical units that are supported.

  Character device drivers will set this field to zero.

On return, the command must also set the status word in the *Request header.*

If the device driver determines that it cannot initialize the device and if it wants to abort the initialization procedure, it must return the following:

Value	Length	Field
00H	BYTE	*Number of units*
0000H	WORD	*Offset ending code address*
0000H	WORD	*Offset ending data address*

Note that the device driver does not indicate failure to initialize with an error return code in the *Status* field of the request packet header.

In the case where the device driver contains multiple device headers, it should remove any unwanted device headers from the list of device headers before those device headers are called for initialization.

To remove the first device header called for initialization means that the device driver should return as if initialization succeeded; then, if a device listed in one of the next headers succeeds, its header information can be copied to the first header (adjusting the linkage to exclude the device header just copied but not destroying the header just copied). Otherwise, if no device is to be supported, the last device header may indicate zero offsets for the code and data segments.

At initialization time, the device driver runs as a thread under a protected-mode process at application level, with I/O privilege. Device drivers that were installed via the **device=** command in the *config.sys* file may issue certain MS OS/2 dynamic-link system calls (see Chapter 2 for a list of these calls). Both base and installable device drivers may use **DevHlp** services that are permitted during initialization.

Media Check: Check the Media (Command Code 1)

Purpose:

Command Code 1 determines the state of the media.

Format of Request Packet:

```
+-------------------------------------+
| 13-BYTE  Request header             |
+-------------------------------------+
| BYTE     Media descriptor           |
+-------------------------------------+
| BYTE     Return code                |
+-------------------------------------+
| DWORD    Return pointer to previous |
|          volume ID if supported     |
+-------------------------------------+
```

Remarks:

On entry, the request packet contains the media descriptor byte:

```
  7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+
| 1 | 1 | 1 | 1 | 1 | x | y | z |
+---+---+---+---+---+---+---+---+
```

where:

Bit	Meaning
0	$z = 1$ means double-sided
1	$y = 1$ means eight sectors/track
2	$x = 1$ means removable

The device driver must perform the following actions:

- Set the status word in the *Request header*.
- Set the return code, with one of the following values:

Value	Meaning
–1	Media has been changed
0	Unsure if media has been changed
1	Media unchanged

Build BPB: Build BIOS Parameter Block (Command Code 2)

Purpose:

Command Code 2 builds the BIOS parameter block, which is requested when the media has changed or when the media type is uncertain.

Format of Request Packet:

13-BYTE	Request header
BYTE	Media descriptor
DWORD	Transfer address
DWORD	Pointer to the BPB table
BYTE	Drive number

where:

Remarks:

On entry, the request packet contains the following fields:

- The *Media descriptor* byte is set.
- The *Transfer address* is set to a buffer containing the boot sector to read from the medium in the drive.

The device driver must perform the following actions:

- Set the *Pointer to the BPB table.*
- Set the status word in the *Request header.*

Read or Write: Perform I/O to Device (Command Codes 4, 8, 9)

Purpose:

Command Code 4 reads from a device; Command Code 8 writes to the device; and Command Code 9 writes, with verify, to the device.

Format of Request Packet:

13-BYTE	Request header
BYTE	Media descriptor
DWORD	Transfer address
WORD	Byte/sector count
DWORD	Starting sector number for block device
WORD	Reserved

where:

Remarks:

On entry, the request packet contains the following fields:

- The *Media descriptor* byte is set.
- The *Transfer address* is set to a buffer.
- The *Byte/sector count* (number of bytes or sectors to transfer) is set.
- The *Starting sector number* is set for the block device.

The device driver must perform the following actions:

- Perform the requested function.
- Set the actual number of sectors or bytes transferred.
- Set the status word in the *Request header.*

Note that the Dword transfer address in the request packet is a locked 32-bit physical address. The device driver can pass it to the **DevHlp** function **PhysToVirt** to obtain a virtual address for the current mode. The device driver does not need to unlock the address when the request is completed.

Nondestructive Read No Wait: Nondestructive Input (Command Code 5)

Purpose:

Command Code 5 reads a character from the buffer but does not remove it.

Format of Request Packet:

```
+-----------------------------------+
| 13-BYTE  Request header           |
+-----------------------------------+
| BYTE     Returned character       |
+-----------------------------------+
```

Remarks:

The device driver must perform the following actions:

- Return a byte from the device.
- Set the status word in the *Request header*.

Status: Input or Output Status (Command Codes 6, 10)

Purpose:

Command Code 6 determines the input status on character devices, and Command Code 10 determines their output status.

Format of Request Packet:

```
+-----------------------------------+
| 13-BYTE    Request header         |
+-----------------------------------+
```

Remarks:

The device driver must perform the following actions:

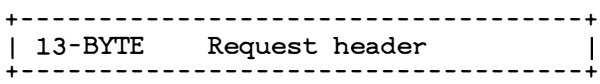
- Perform the requested function.
- Set the busy bit.
- Set the status word in the *Request header*.

Flush: Input or Output Flush (Command Codes 7, 11)

Purpose:

Command Code 7 flushes all pending requests and Command Code 11 terminates them.

Format of Request Packet:



Remarks:

The device driver must perform the following actions:

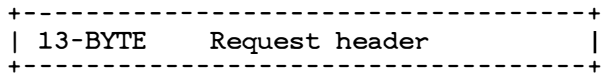
- Perform the requested function.
- Set the status word in the *Request header*.

Open or Close: Open/Close Device (Command Codes 13, 14)

Purpose:

Command Code 13 opens the device, and Command Code 14 closes it.

Format of Request Packet:



Remarks:

The device driver must perform the following actions:

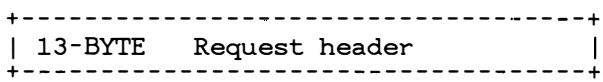
- Perform the requested function.
- Set the status word in the *Request header*.

Removable Media: Check for Removable Media (Command Code 15)

Purpose:

Command code 15 checks for removable media.

Format of Request Packet:



Remarks:

The device driver must perform the following actions:

- Set the busy bit of the status word if the media is nonremovable; clear it if the media is removable.
- Set the status word in the *Request header*.

Generic IOCtl: I/O Control for Devices (Command Code 16)

Purpose:

Command Code 16 sends I/O control commands to a device.

Format of Request Packet:

13-BYTE	Request header
BYTE	Function category
BYTE	Function code
DWORD	Pointer to parameter buffer
DWORD	Pointer to data buffer

Remarks:

On entry, the request packet contains the following fields:

- The *Function category* is set.
- The *Function code* is set.
- The *Pointer to parameter buffer* is set as a virtual address.
- The *Pointer to data buffer* is set as a virtual address.

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the *Request header*.

If the function cannot be performed immediately, the device driver must perform the following actions:

- Use the **DevHlp** function **Lock** to lock the memory segment
- Convert the address of the parameter and data buffers to 32-bit physical addresses using the **DevHlp** function **VirtToPhys**
- Sort the addresses back in the request packet
- Convert the addresses to virtual addresses by using the **DevHlp** function **PhysToVirt** whenever the device driver needs to use the addresses
- Unlock the memory segment using the **DevHlp** function **UnLock**.

This conversion ensures that the correct virtual address is used for the current mode, whether real or protected.

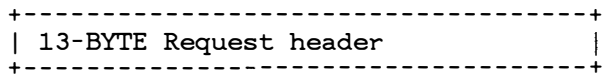
For more information about IOCtl commands, see Chapter 4, "Generic IOCtl Commands."

Reset Media: Reset Uncertain Media Condition (Command Code 17)

Purpose:

Command Code 17 resets the "uncertain media" error condition and allows MS OS/2 to identify media.

Format of Request Packet:



Remarks:

On entry, the unit code identifies the unit number to be reset.

Before this command was implemented, the device driver returned the error condition "uncertain media."

The device driver must perform the following action:

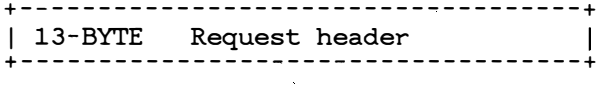
- Set the status word in the *Request header*.

Logical Drive: Get/Set Logical Drive Mapping (Command Codes 18, 19)

Purpose:

Command Code 18 *gets* which logical drive is currently mapped onto a particular unit. Command Code 19 *sets* which logical drive is currently mapped onto a particular unit.

Format of Request Packet:



Remarks:

On entry, the unit code in the request header contains the unit number of the drive for which this operation is to be performed. The device driver must perform the following actions:

- For Get, it must return the logical drive that is mapped onto the physical drive indicated by the unit code in the *Request header*.

  For Set, it must map the logical drive indicated by the unit code in the *Request header* onto the physical drive that contains the mapping of logical drives.
- The logical drive is returned in the unit code field of the *Request header*. If there is only one logical drive mapped onto the physical drive, the device driver should set this unit code field to zero.
- Set the status word in the *Request header*.

Deinstall: Terminate the Device Driver (Command Code 20)

Purpose:

Command Code 20 terminates the device driver and allows it to relinquish its interrupt vectors and allocated memory.

Format of Request Packet:

```
+-----------------------------------+
| 13-BYTE   Request header          |
+-----------------------------------+
```

Remarks:

The device driver must perform the following actions:

- Release any allocated physical memory.
- Unset any hardware vectors that it had claimed.
- If the device driver has a ROM BIOS interrupt handler, it cannot reset the vector. Instead, it must preserve the real mode vector chain by doing a JMP to the previous handler.
- Perform any other cleanup.
- Clear the error bit in the status field in the request header to indicate successful deinstall.

If the device driver determines that it cannot or will not abort, it should

- Set the error bit in the status field in the request header and set the error code to 03H, "unknown command."

If the device driver refuses to deinstall, the second device driver will not be allowed to install.

Port Access: Grant Port Access (Command Code 21)

Purpose:

Command Code 21 grants port access to a user process for IOPL.

Format of Request Packet:

```
+-----------------------------------+
| 13-BYTE   Request header          |
+-----------------------------------+
| BYTE      Type of access          |
+-----------------------------------+
| WORD      FirstPort               |
+-----------------------------------+
| WORD      LastPort                |
+-----------------------------------+
| WORD      Port status             |
+-----------------------------------+
```

Remarks:

On entry, the request packet contains the following fields:

- The *Type of access* field indicates a request for or a release of access to the port:

Value	Meaning
0	Request access
1	Release access

- The *FirstPort* field specifies either the starting (low-end) number in a contiguous range, or a single port.

- The *LastPort* field specifies either the ending (high-end) number in a contiguous range, or a single port. If only one port is being used, *FirstPort* and *LastPort* must be set to that port's number.

The device driver will only be called for those ports for which "exclusive-use-only" with the **DevHlp** function, **PortUsage**.

The device driver must perform the following actions:

- If access is allowed/released, call the **DevHlp** function **GrantPortAccess** to give or remove the user process access.
- Set the *Port status* word in the request packet.

Value	Meaning
0000H	Access allowed
0FFFFH	Access denied

- Set the status word in the *Request header*.

Partitionable Fixed Disks: Get Number of Fixed Disks (Command Code 22)

Purpose:

Command Code 22 is used by the system to get the number of partitionable fixed disks supported by the device driver.

Format of Request Packet:

13-BYTE	Request header
BYTE	Count
WORD	Reserved
WORD	Reserved

Remarks:

The device driver must perform the following actions:

- Set the *Count* field to the one-based number of partitionable fixed disks supported by the device driver.
- Set the *Status* word in the request header.

Command code 22 call allows the Category 09H generic IOCtl commands to be routed to the correct device driver. This call is not tied to a particular unit that the device driver owns, but is directed to the device driver as a general query of its device support.

Get Fixed Disk Map: Get Fixed Disk/Logical Unit Map (Command Code 23)

Purpose:

Command Code 23 determines which logical units supported by the device driver actually exist on a physical partionable fixed disk.

Format of Request Packet:

13-BYTE	Request header
4 BYTES	Units-supported bit mask
WORD	Reserved = 0
WORD	Reserved = 0

Remarks:

On entry, the request packet contains the following:

- The *Unit* field in the *Request header* contains the zero-based unit number of the physical partitionable fixed disk that is the target of this request. The unit number is relative to the number of partitionable disks that the device driver had reported back to the system in response to Partitionable Fixed Disks (Command Code 22).

The device driver must set the following:

- The *Unit-supported bit mask* is a four-byte bit mask that must be set to indicate which of the device driver's logical units actually exist on the physical partitionable fixed disk that was specified on entry.

A zero means the logical unit does not exist, and a 1 means that it does exist. The first logical unit that the device driver supports is the low-order bit of the first byte (that is, it is zero-based). The bits are used from right-to-left starting at the low-order bit of each following byte. It is possible that all the bits will be zero.

For example, in a particular system, a block device driver supports five units spread over two floppy disk drives and one partitionable fixed disk. Units 0 and 1 map to the floppy disk drives. Units 2, 3, and 4 map to the fixed disk. For the device command, this device driver sets the four-byte bitmap to:

```
0000 0000 0000 0000 0000 0000 0001 1100        (binary)
```

or to:

```
00 00 00 1C                                    (hexadecimal)
```

The physical drive relates only to the drives reported to the operating system via Partitionable Fixed Disks (Command Code 22). It is possible that no logical units exist on a given physical disk because it has not yet been initialized.

Chapter 4
Generic IOCtl Commands

4.1 Introducing the Generic IOCtl Commands

This chapter describes the details of the Generic IOCtl commands given via the Generic IOCtl command (Command Code 16) in Chapter 3, "Device Driver Commands."

The category and function fields in the Request Packet are determined as follows (each code is contained in a byte):

Category code

The category code is bitmapped as follows:

Bit	Meaning
0–6	Reserved
7	0 = MS OS/2-defined 1 = User-defined

Function code

The function code is bitmapped as follows:

Bit	Meaning
0–4	Subfunction code
5	0 = Sends data/commands to device 1 = Queries data/information from device
6	0 = Intercepted by MS OS/2 1 = Passed to driver
7	0 = Return error if unsupported 1 = Ignore if unsupported

Note

The "Sends/Queries" data bit is intended only to make a function set orthogonal. It plays no critical role; some functions may contain both command and query elements. The convention is that such commands are defined as "Sends data."

The list of commands defined for the Generic IOCtl request is as follows:

Category	Function	Description
01H		*Serial Device Control*
	41H	Set Baud Rate
	42H	Set Line Control Register
	43H	Set XON/XOFF characters
	44H	Transmit Immediate
	45H	Set Break Off
	46H	Set Modem Control Register
	4BH	Set Break On
	4CH	Stop Transmit
	4DH	Start Transmit
	4EH	Set I/O Control Behavior
	52H	Set Communications Event Mask
	53H	Set Device Control Information
	61H	Return Current Baud Rate
	62H	Return Line Control Register
	63H	Return Flow Control Characters
	65H	Return Current Line Status
	66H	Return Modem Control Register
	67H	Return Current Modem Status
	68H	Return Number of Characters in Input Queue
	69H	Return Number of Characters in Output Queue
	6BH	Return Communications Status
	6DH	Return Communications Error

	6EH	Return I/O Control Setting Status
	72H	Get Communications Event Mask
	73H	Get Device Control Information
02H		*Reserved*
03H		*Screen/Pointer Draw Control*
	63H/0BH	Get ROM Font Information
	63H/0CH	Get Video Environment
	64H	Get Code Page Information
	6DH	Reserved
	6EH	Reserved
	6FH	Reserved
	70H	Get Screen Buffer Selector
	71H	Release Screen Buffer Selector
	72H	Get Pointer Draw Address
	73H/05H	Set Video Mode
	73H/0DH	Set Video Environment
	73H/10H	Set Character Font
	73H/12H	Set Palette Registers
	74H	Set Code Page Information
04H		*Keyboard Control*
	50H	Set Translate Table
	51H	Set Input Mode
	52H	Set Interim Character Flags
	53H	Set Shift State
	54H	Set Typamatic Rate and Delay
	55H	Set Foreground Screen Group
	56H	Set Session Manager Hot Key
	57H	Set Pause Semaphore Handle
	71H	Get Input Mode
	72H	Get Interim Character Flags
	73H	Get Shift State
	74H	Read Character Data Record(s)

	75H	Peek Character Data Record(s)
	76H	Get Session Manager Hot Key
	77H	Get Keyboard Type
05H		*Printer Control*
	42H	Set Frame Control
	44H	Set Infinite Retry
	46H	Initialize Printer
	62H	Get Frame Control
	64H	Get Infinite Retry
	65H	Get Printer Status
06H		*Light Pen Control*
07H		*Mouse Control*
	50H	Allow Pointer Drawing after Screen Switch
	51H	Update Screen Display Mode
	52H	Screen Switcher Notification
	53H	Set New Scaling Factors
	54H	Set New Event Mask
	55H	Set System Hot Key Button
	56H	Set Pointer Shape
	57H	Unmark Collision Area
	58H	Mark Collision Area
	59H	Set Pointer Screen Position
	5AH	Set Protected-Mode Pointer-Draw Address
	5BH	Set Real-Mode Pointer-Draw Address
	5CH	Set Mouse Status Flags
	60H	Get Number of Buttons
	61H	Get Number of Mickeys/centimeter
	62H	Get Device Status Flags
	63H	Read Event Queue
	64H	Get Event Queue Status
	65H	Get Current Event Mask
	66H	Get Current Scaling Factors

	67H	Get Pointer Screen Position
	68H	Get Pointer Shape
	69H	Get System Hot Key Button
08H		*Disk/Diskette Control*
	00H	Lock Drive
	01H	Unlock Drive
	02H	Redetermine Media
	03H	Set Logical Map
	20H	Block Removable
	21H	Get Logical Map
	22H	Reserved
	43H	Set Device Parameters
	44H	Write Track
	45H	Format and Verify Track on a Drive
	5FH	Reserved
	63H	Get Device Parameters
	64H	Read Track
	65H	Verify Track
09H		*Physical Disk Control*
	00H	Lock Physical Drive
	01H	Unlock Physical Drive
	44H	Physical Write Track
	63H	Get Physical Device Parameters
	64H	Physical Read Track
	65H	Physical Verify Track
0AH		*Character Monitor Control*
	40H	Register Monitor
0BH		*General Device Control*
	01H	Flush Input Buffer
	02H	Flush Output Buffer

60H Query Monitor Support

The following sections describe the formats of the IOCtl commands.

4.2 Serial Device Control IOCtl Commands (Category 01H)

This section describes the serial device control subfunctions within Category 01H of the Generic IOCtl commands. The following lists show the serial device subfunctions by category; following these lists, the subfunctions are described in numerical order.

Line Characteristics

Function	Description
41H	Set Baud Rate
42H	Set Line Control Register
46H	Set Modem Control Register
61H	Return Current Baud Rate
62H	Return Line Control Register
65H	Return Current Line Status
66H	Return Modem Control Register
67H	Return Current Modem Status

Manual XON/XOFF (Flow Control) Processing

Function	Description
44H	Transmit Immediate
4CH	Stop Transmit
4DH	Start Transmit

Automatic XON/XOFF (Flow Control) Processing

Function	Description
43H	Set Flow Control Characters

4EH	Set I/O Control Behavior
63H	Return Flow Control Characters
6EH	Return I/O Control Setting

Break Processing

Function	Description
45H	Set Break Off
4BH	Set Break On

Device Driver I/O Queue Management

Function	Description
68H	Return Number of Characters in Input Queue
69H	Return Number of Characters in Output Queue

Polled Events

Function	Description
52H	Set Communications Event Mask
6BH	Return Communications Status
6DH	Return Communications Error
72H	Return Communications Event Mask

General Device Control Block Parameter Access

Function	Description
53H	Set Device Control Block Information
73H	Return Device Control Block Information

Function 41H: Set Baud Rate

Purpose:

Function 41H sets the baud (bit) rate of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| WORD   Bit rate                   |
+-----------------------------------+
```

where:

Bit rate is a Word that specifies the actual bit rate. The valid range of *Bit rate* values is:

$50 \leq$ *Bit rate* $\leq$ 19,200

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

None.

Returns:

If the call is made with the *Bit rate* out of range, a "General failure" error is reported.

Function 42H: Set Line Control Register

Purpose:

Function 42H sets the line control register (stop bits, parity, data bits) of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Data bits                  |
|-----------------------------------|
| BYTE   Parity                     |
|-----------------------------------|
| BYTE   Stop bits                  |
+-----------------------------------+
```

where:

Data bits has one of the following values:

Value	Meaning
00H–04H	Reserved
05H	5 data bits
06H	6 data bits
07H	7 data bits
08H	8 data bits
09H–0FFH	Reserved

Parity is one of the following:

Value	Meaning
00H	No parity
01H	Odd parity
02H	Even parity

03H	Mark parity
04H	Space parity
05H–0FFH	Reserved

Stop bits has one of the following values:

Value	Meaning
00H	1 stop bit
01H	1.5 stop bits
02H	2 stop bits
03H–0FFH	Reserved

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

The three bytes of input are used to construct the actual line control register value used to set the hardware.

Returns:

Any invalid input causes the function to return a "General failure" error.

Function 43H: Set Flow Control Characters

Purpose:

Function 43H sets XON/XOFF (flow control) characters for a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   XON character              |
|-----------------------------------|
| BYTE   XOFF character             |
+-----------------------------------+
```

where:

The *XON/XOFF character* values are used for automatic XON/XOFF flow control. The default values are as follows:

Value	Meaning
XON	01H (DC1)
XOFF	03H (DC3)

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

Setting the XON and XOFF characters does not mean that XON/XOFF flow control is enabled. The two bits that enable and disable input and output flow control are in the *Flags2* byte of the device control block information, which is controlled by the Category 01H IOCtl **Functions 53H** and **73H**.

Function 44H: Transmit Immediate

Purpose:

Function 44H transmits a byte immediately to a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Character to be transmitted |
+-----------------------------------+
```

where:

Character to be transmitted is a Byte that specifies the XON or XOFF character to be transmitted, usually manually, by this function. Characters sent by **Function 44H** will bypass the device driver buffer.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

None.

Function 45H: Set Break Off

Purpose:

Function 45H sets Break off and resumes transmitting the data in the output queue.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Communications error       |
+-----------------------------------+
```

where:

Communications error is a Word that contains the communications error. On error, this Word will be nonzero. *Communications error* is bitmapped as follows:

Bit	Meaning
0	The requested mode is not supported, or the communication handle is invalid. If this bit is set, this is the only valid error.
1	Receive queue overflow.
2	A character was not read from the hardware before the next character arrived. The character was lost.
3	The hardware detected a parity error.
4	The hardware detected a framing error.
5	The hardware detected a Break condition.
6	"Clear to send" (CTS) time-out.

7 "Data set ready" (DSR) time-out.

8 "Receive line signal detect" (RLSD) time-out.

9 Transmit queue was full while trying to queue a character.

10–15 Reserved

Remarks:

None.

Function 46H: Set Modem Control Register

Purpose:

Function 46H sets the modem control register of a serial device.

Parameter Packet Format:

```
+-------------------------------------+
| WORD   Modem control register       |
+-------------------------------------+
```

where:

Modem control register is a Word that specifies the on or off setting of the DTR and RTS lines. The high byte contains a mask of the bits to turn off. The low byte contains a mask of the bits to turn on. The possible values for the *Modem control register* are:

Value	Meaning
0FF01H	Set DTR
0FE00H	Clear DTR
0FD02H	Set RTS
0FD00H	Clear RTS

Data Packet Format:

```
+-----------------------------------+
| WORD   Communications error       |
+-----------------------------------+
```

where:

Communications error is a Word that contains the communications error. On error, this word will be nonzero. *Communications error* is bitmapped as follows:

Bit	Meaning

0	The requested mode is not supported, or the communication handle is invalid. If this bit is set, this is the only valid error.
1	Receive queue overflow.
2	A character was not read from the hardware before the next character arrived. The character was lost.
3	The hardware detected a parity error.
4	The hardware detected a framing error.
5	The hardware detected a Break condition.
6	"Clear to send" (CTS) time-out.
7	"Data set ready" (DSR) time-out.
8	"Receive line signal detect" (RLSD) time-out.
9	Transmit queue was full while trying to queue a character.
10–15	Reserved

Remarks:

Following is a definition of the modem control register's hardware:

Bit	Meaning
0	Data terminal ready
1	Request to send
2	Output 1
3	Output 2
4	Loop-back mode
5–7	Undefined

Function 4BH: Set Break On

Purpose:

Function 4BH sets Break on.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Communications error       |
+-----------------------------------+
```

where:

Communications error is a Word that contains the communications error. On error, this Word will be nonzero. *Communications error* is bitmapped as follows:

Bit	Meaning
0	The requested mode is not supported, or the communication handle is invalid. If this bit is set, this is the only valid error.
1	Receive queue overflow.
2	A character was not read from the hardware before the next character arrived. The character was lost.
3	The hardware detected a parity error.
4	The hardware detected a framing error.
5	The hardware detected a Break condition.
6	"Clear to send" (CTS) time-out.
7	"Data set ready" (DSR) time-out.

8 "Receive line signal detect" (RLSD) time-out.

9 Transmit queue was full while trying to queue a character.

10–15 Reserved

Remarks:

Function 4BH signals the device driver to generate the Break signal as soon as possible. The function does not wait for the transmitter-holding register and shift register to empty.

When the Break function is issued, the Break signal is transmitted even if transmission is currently suspended due to XOFF.

Function 4CH: Stop Transmit

Purpose:

Function 4CH causes the serial line to behave as if XOFF were received (it stops transmission of data).

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

This function halts data transmission. However, characters transmitted by this function will *not* be prohibited by it.

Function 4DH: Start Transmit

Purpose:

Function 4DH causes the serial line to behave as if XON were received (it starts or resumes transmission of data).

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

This function resumes data transmission.

Function 4EH: Set I/O Control Behavior

Purpose:

Function 4EH sets input and output XON/XOFF behavior for a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| WORD   Input/Output control       |
+-----------------------------------+
```

where:

Input/Output control is a Word that specifies XON/XOFF control. The high byte contains output XON/XOFF control, and the low byte contains input XON/XOFF control. The possible values for this word are as follows:

Value	Meaning
00*xx*H	Disable output flow control
01*xx*H	Enable output flow control
*xx*00H	Disable input flow control
*xx*01H	Enable input flow control

For example, *Input/Output control* set to 0101H means to enable both output and input flow control.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

When XON and XOFF flow control during transmission is enabled, the communications device driver stops transmitting when it receives an XOFF, and resume transmission when it receives an XON.

When XON and XOFF flow control during reception is enabled, the *COM* device driver transmits an XOFF when its input queue gets close to full, and an XON when its input queue gets close to empty. After the XOFF is sent, the *COM* device driver sends no characters until the input queue gets close to empty. This is to accommodate those systems that interpret the first character received after an XOFF as an XON, regardless of what the character really is.

Function 52H: Set Communications Event Mask

Purpose:

Function 52H sets the communications event mask of a serial device.

Parameter Packet Format:

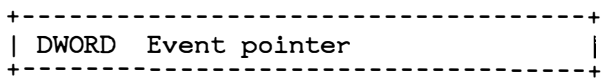
```
+-----------------------------------+
| WORD   Event enable mask          |
+-----------------------------------+
```

where:

Event enable mask is a Word that specifies which events are to be enabled. Any combination of bits can be set. *Event enable mask* is bit-mapped as follows:

Bit	Meaning
0	Set when any character is received and placed in the receive queue.
1	Set when the event character is received and placed in the receive queue. The event character is specified in the device's control block.
2	Set when the last character in the transmit queue is sent.
3	Set when the "Clear to send" (CTS) signal changes state.
4	Set when the "Date set ready" (DSR) signal changes state.
5	Set when the "Receive line signal detect" (RLSD) signal changes state.
6	Set when a Break is detected.
7	Set when a line status error occurs: the errors are parity, framing, and overrun.
8	Set when a ring indicator is detected.

Data Packet Format:



where:

Event pointer is a Dword that specifies the virtual address, in the user's LDT, of the event word. Each bit in the returned event mask specifies whether a given event has occurred. A bit is 1 if the event has occurred.

Remarks:

Function 52H enables the communications device event mask and returns a virtual pointer to the event word. The bits of the *Event enable mask* parameter define which events are to be enabled. The return value points to the current state of the event mask.

The communications device driver uses the **DevHlp** function, **PhyToUVirt**, to return a selector:offset or segment:offset address of the Event word. The Event word is located in the communication device driver's data segment.

Function 53H: Set Device Control Block Information

Purpose:

Function 53H sets information for the device control block (DCB).

Parameter Packet Format:

```
+-----------------------------------+
| WORD   RLSD timeout               |
|-----------------------------------|
| WORD   CTS timeout                |
|-----------------------------------|
| WORD   DSR timeout                |
|-----------------------------------|
| BYTE   Flags1                     |
|-----------------------------------|
| BYTE   Flags2                     |
|-----------------------------------|
| BYTE   XON threshold value        |
|-----------------------------------|
| BYTE   XOFF threshold value       |
|-----------------------------------|
| BYTE   Error replacement character |
|-----------------------------------|
| BYTE   End of input character     |
|-----------------------------------|
| BYTE   Event generating character |
+-----------------------------------+
```

where:

RLSD timeout is the amount of time, in milliseconds, to wait for RLSD to be set.

CTS timeout is the amount of time, in milliseconds, to wait for CTS to be set.

DSR timeout is the amount of time, in milliseconds, to wait for DSR to be set.

Flags1: The bits in the *Flags1* byte are defined as follows:

Bit	Meaning
0	Binary mode (ignore EOF character)
1	Disable RTS
2	Enable sensitivity to hardware parity checking
3	Output handshaking using CTS
4	Output handshaking using DSR
5–6	Reserved
7	Disable DTR

Flags2: The bits in the *Flags2* byte are defined as follows:

Bit	Meaning
0	Output-automatic XON/XOFF active
1	Input-automatic XON/XOFF active
2	Error-replacement character active
3	Null stripping (remove null bytes)
4	Character event monitoring active
5	Input handshaking using DTR
6	Input handshaking using RTS
7	Reserved

XON threshold value is the input queue's threshold level to transmit XON.

XOFF threshold value is the input queue's threshold level to transmit XOFF.

The XOFF level must be set less than XON. Trying to set XOFF greater than XON will return an error condition.

The device driver supplies default levels, but applications can reset these values during operation.

Error replacement character is a Byte that is placed in an input data stream when a hardware error is detected (parity, framing, etc).

The *End of input character* (EOF) indicates the end of a data stream. When it encounters an EOF, the device driver will discard all subsequent characters. No data may be read past the EOF character.

The *Event generating character* "watches" for a particular character in the data stream. When it "recognizes" the character, the appropriate bit is set in the event word.

Data Packet Format:

```
+--------------------------------+
| DWORD  Set to zero             |
+--------------------------------+
```

Remarks:

None.

Function 61H: Return Current Baud Rate

Purpose:

Function 61H returns the current baud (bit) rate of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Bit rate                   |
+-----------------------------------+
```

where:

Bit rate is a Word that specifies the actual baud rate in bits/second. The valid range of *Bit rate* values is as follows:

$50 \leq$ *Bit rate* $\leq$ 19,200

Remarks:

None.

Function 62H: Return Line Control Register

Purpose:

Function 62H returns the line control register (stop bits, parity, data bits) of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| BYTE   Line control register value |
|-----------------------------------|
| BYTE   Data bits                  |
|-----------------------------------|
| BYTE   Parity                     |
|-----------------------------------|
| BYTE   Stop bits                  |
+-----------------------------------+
```

where:

Line control register value is the byte contained in the hardware line control register.

Data bits is a Byte that has one of the following values:

Value	Meaning
00H–04H	Reserved
05H	5 data bits
06H	6 data bits
07H	7 data bits

08H	8 data bits
09H–0FFH	Reserved

Parity is one of the following:

Value	Meaning
00H	No parity
01H	Odd parity
02H	Even parity
03H	Mark parity
04H	Space parity
05H–0FFH	Reserved

Stop bits has one of the following values:

Value	Meaning
00H	1 stop bit
01H	1.5 stop bits
02H	2 stop bits
03H–0FFH	Reserved

Remarks:

None.

Function 63H: Return Flow Control Characters

Purpose:

Function 63H returns XON/XOFF (flow control) characters for a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| BYTE   XON character              |
|-----------------------------------|
| BYTE   XOFF character             |
+-----------------------------------+
```

where:

The *XON/XOFF character* Bytes are used for automatic XON/XOFF flow control. Their default values are as follows:

Value	Meaning
XON	01H (DC1)
XOFF	03H (DC3)

Remarks:

None.

Function 65H: Return Current Line Status

Purpose:

Function 65H returns the status of the current line of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| BYTE   Line status register value |
+-----------------------------------+
```

where:

Line status register value is a Byte that is bitmapped as follows:

Bit	Meaning
0	Data ready
1	Overrun error
2	Parity error
3	Framing error
4	Break interrupt
5	TX holding register empty
6	TX shift register empty
7	Reserved

Remarks:

None.

Function 66H: Return Modem Control Register

Purpose:

Function 66H returns the modem control register of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+--------------------------------------+
| BYTE   Modem control register        |
+--------------------------------------+
```

where:

Modem control register is a Byte that specifies the state of the hardware modem control register. The mask used in the Set function, Category 01H, IOCtl **Function 46H**, is not used for this function. Following is a definition of the modem control register's hardware:

Bit	Meaning
0	Data terminal ready
1	Request to send
2	Output 1
3	Output 2
4	Loop-back mode
5–7	Undefined

Remarks:

None.

Function 67H: Return Current Modem Status

Purpose:

Function 67H returns the current status of the modem of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| BYTE   Modem status register      |
+-----------------------------------+
```

where:

Modem status register is a Byte with the following bitmap:

Bit	Meaning
0	Delta clear to send
1	Delta data set ready
2	Trailing edge of ring
3	Delta receiver line signal
4	Clear to send (CTS)
5	Data set ready (DSR)
6	Ring indicator
7	Receiver line signal detect (RLSD)

Remarks:

None.

Function 68H: Return Number of Characters in Input Queue

Purpose:

Function 68H returns the number of characters in the input queue of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Queue number of characters |
+-----------------------------------+
```

where:

Queue number of character is a Word that specifies the number of characters in the input queue.

Remarks:

None.

Function 69H: Return Number of Characters in Output Queue

Purpose:

Function 69H returns the number of characters in the output queue of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Queue number of characters |
+-----------------------------------+
```

where:

Queue number of characters is a binary integer defining the number of characters in the output queue.

Remarks:

None.

Function 6BH: Return Communications Status

Purpose:

Function 6BH returns the communications status of a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| BYTE   COM port status            |
+-----------------------------------+
```

where:

The bits in the *COM port status* Byte are described as follows:

Bit	Meaning
0	TX is waiting for CTS to be asserted
1	TX is waiting for DSR to be asserted
2	TX is waiting for RLSD to be asserted
3	TX is waiting because an XOFF was received
4	TX is waiting because an XOFF was transmitted
5	EOF character was received
6	Character waiting to transmit immediate
7	Reserved

Remarks:

None.

Function 6DH: Return Communications Error

Purpose:

Function 6DH returns a communications error, then gets and clears the communication device's error value in the device control block

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Communications error       |
+-----------------------------------+
```

where:

Communications error is a Word that contains the communications error. On error, this word will be nonzero. *Communications error* is bitmapped as follows:

Bit	Meaning
0	The requested mode is not supported, or the communication handle is invalid. If this bit is set, this is the only valid error.
1	Receive queue overflow.
2	A character was not read from the hardware before the next character arrived. The character was lost.
3	The hardware detected a parity error.
4	The hardware detected a framing error.
5	The hardware detected a Break condition.
6	"Clear to send" (CTS) time-out.

7	"Data set ready" (DSR) time-out.
8	"Receive line signal detect" (RLSD) time-out.
9	Transmit queue was full while trying to queue a character.
10–15	Reserved

Remarks:

None.

Function 6EH: Return I/O Control Setting

Purpose:

Function 6EH returns input and output XON/XOFF control setting for a serial device.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Input/Output control       |
+-----------------------------------+
```

where:

Input/Output Control is a Word that specifies XON/XOFF control. The high byte contains output XON/XOFF control, and the low byte contains input XON/XOFF control. The possible values for this word are as follows:

Value	Meaning
00*xx*H	Output flow control disabled
01*xx*H	Output flow control enabled
*xx*00H	Input flow control disabled
*xx*01H	Input flow control enabled

Remarks:

When XON and XOFF flow control during transmission is enabled, the communications device driver stops transmitting when it receives an XOFF, and resumes transmission when it receives an XON.

When XON and XOFF flow control during reception is enabled, the *COM* device driver transmits an XOFF when its input queue gets close to full, and an XON when its input queue gets close to empty. After the XOFF is sent, the *COM* device driver sends no characters until the input queue gets close to empty. This is to accommodate those systems that interpret the first character received after an XOFF as an XON, regardless of what the character really is.

Function 72H: Return Communications Event Mask

Purpose:

Function 72H returns the communications event mask, then retrieves and clears the *COM* device event mask.

Parameter Packet Format:

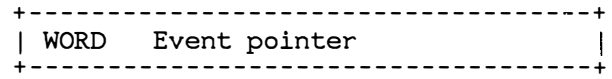
```
+-----------------------------------+
| WORD   Event enable mask          |
+-----------------------------------+
```

where:

Event enable mask contains a short integer value whose bits specify which events are to be enabled. Any combination of the following bits can be set:

Bit	Meaning
0	Set when any character is received and placed in the receive queue.
1	Set when the event character is received and placed in the receive queue. The event character is specified in the device's control block.
2	Set when the last character in the transmit queue is sent.
3	Set when the "Clear to send" (CTS) signal changes state.
4	Set when the "Data set ready" (DSR) signal changes state.
5	Set when the "Receive line signal detect" (RLSD) signal changes state.
6	Set when a Break is detected.
7	Set when a line status error occurs: the errors are parity, framing, and overrun.
8	Set when a ring indicator is detected.

Data Packet Format:



where:

Event pointer is a Word that specifies the virtual address, in the user's LDT, of the event word. Each bit in the returned event mask specifies whether a given event has occurred. A bit is 1 if the event has occurred.

Remarks:

The communications device driver uses the **DevHlp** function, **PhyToUVirt**, to return a selector:offset or segment:offset address of the Event word. The Event word is located in the communication device driver's data segment.

This function must be used to prevent the loss of an event.

Function 73H: Return Device Control Block Information

Purpose:

Function 73H returns device control block information.

Parameter Packet Format:

DWORD	Set to zero

Data Packet Format:

WORD	RLSD timeout
WORD	CTS timeout
WORD	DSR timeout
BYTE	Flags1
BYTE	Flags2
BYTE	XON threshold value
BYTE	XOFF threshold value
BYTE	Error replacement character
BYTE	End of input character
BYTE	Event generating character

where:

RLSD timeout is the amount of time, in milliseconds, to wait for RLSD to be set.

CTS timeout is the amount of time, in milliseconds, to wait for CTS to be set.

DSR timeout is the amount of time, in milliseconds, to wait for DSR to be set.

The bits in the *Flags1* byte are defined as follows:

Bit	Meaning
0	Binary mode (ignore EOF character)
1	Disable RTS
2	Enable sensitivity to hardware parity checking
3	Output handshaking using CTS
4	Output handshaking using DSR
5	Reserved
6	Reserved
7	Disable DTR

The bits in the *Flags2* byte are defined as follows:

Bit	Meaning
0	Output-automatic XON/XOFF active
1	Input-automatic XON/XOFF active
2	Error-replacement character active
3	Null stripping (remove null bytes)
4	Character event monitoring active
5	Input handshaking using DTR
6	Input handshaking using RTS
7	Reserved

XON threshold value is the input queue's threshold level to transmit XON.

XOFF threshold value is the input queue's threshold level to transmit XOFF.

The XOFF level must be set less than XON. Setting XOFF greater than XON returns an error condition.

The device driver supplies default levels, but applications can reset these values during operation.

The *error replacement character* is placed in an input data stream when a hardware error is detected (parity, framing, etc).

The *end of input character* (EOF) indicates the end of a data stream. When it encounters an EOF, the device driver will discard all subsequent characters. No data may be read past the EOF character.

The *event generating character* "watches" for a particular character in the data stream. When it recognizes the character, the appropriate bit is set in the event word.

Remarks:

None.

4.3 Screen/Pointer Draw IOCtl Commands (Category 03H)

This section describes the screen and pointer-draw control subfunctions within Category 03H of the Generic IOCtl commands. These subfunctions are listed as follows:

Function	Description
63H/0BH	Get ROM Font Information
63H/0CH	Get Video Environment
64H	Get Code Page Information
6DH	Reserved
6EH	Reserved
6FH	Reserved
70H	Get Screen Buffer Selector
71H	Release Screen Buffer Selector
72H	Get Pointer-Draw Address
73H/05H	Set Video Mode
73H/0DH	Set Video Environment
73H/10H	Set Character Font
73H/12H	Set Palette Registers
74H	Set Code Page Information

Function 63H/0BH: Get ROM Font Information

Purpose:

Function 63H , Type 0BH, which is supported by the screen device driver, returns information about the current ROM font.

Parameter Packet Format:

```
+--------------------------------------+
| WORD   Parameter block length        |
|--------------------------------------|
| WORD   Request type = 000BH          |
|--------------------------------------|
| WORD   Size of data block            |
+--------------------------------------+
```

where:

Parameter block length is a word specifying the length of the parameter packet in bytes.

Request type is the type of request for this function that must be set to 000BH.

Size of data block is a word containing the length, in bytes, of the data packet that contains the ROM font information.

Data Packet Format:

```
+--------------------------------------+
| n BYTES  ROM font info buffer        |
+--------------------------------------+
```

where:

ROM font info buffer is an *n*-byte structure containing the ROM font information. The length of this structure is defined in the parameter packet.

Remarks:

This function is supported by the screen device driver.

Function 63H/0CH: Get Video Environment

Purpose:

Function 63H, **Type 0CH**, which is supported by the screen device driver, returns information about the current video environment.

Parameter Packet Format:

```
+--------------------------------------+
| WORD   Parameter block length        |
|--------------------------------------|
| WORD   Request type = 000CH          |
|--------------------------------------|
| WORD   Size of data block            |
|--------------------------------------|
| WORD   Flags                         |
+--------------------------------------+
```

where:

Parameter block length is a word specifying the length of the parameter packet in bytes.

Request type is the type of request for this function that must be set to 000CH.

Size of data block is a word containing the length, in bytes, of the data packet that contains the video environment buffer.

Data Packet Format:

```
+--------------------------------------+
| n BYTES  Video environment buffer    |
+--------------------------------------+
```

where:

Video environment buffer is an *n*-byte structure containing the video environment buffer.

Remarks:

This function is supported by the screen device driver.

Function 64H: Get Code Page Information

Purpose:

Function 64H, which is supported by the screen device driver, returns information about the current code page.

Parameter Packet Format:

```
+---------------------------------------+
| DWORD   Set to zero                   |
+---------------------------------------+
```

Data Packet Format:

```
+---------------------------------------+
| WORD   Length of data block           |
|---------------------------------------|
| WORD   Number of code page ids        |
|---------------------------------------|
| WORD   First code page id             |
+---------------------------------------+
                 :
                 :
+---------------------------------------+
| WORD   Nth code page id               |
|---------------------------------------|
| n BYTES  Code page path specifier     |
+---------------------------------------+
```

where:

Length of data block specifies the total length, in bytes, of the data packet.

Number of code page ids specifies the number of code page identifiers that are listed in the data packet.

First code page id through the *Nth code page id* are code page identifiers that are used for the code page file which were specified in the *config.sys* file.

Code page path specifier is an *n* byte structure specifying the path to the code page file which were specified in the *config.sys* file.

Remarks:

There is no parameter packet for this function.

This function is supported by the screen device driver.

Function 70H: Get Screen Buffer Selector

Purpose:

Function 70H, which is supported by the screen device driver, returns a protected-mode selector that can access the physical screen buffer.

Parameter Packet Format:

```
+-------------------------------------------------+
| DWORD  32-bit physical screen buffer address    |
|-------------------------------------------------|
| WORD   Length of screen buffer                  |
+-------------------------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD   Screen buffer selector     |
+-----------------------------------+
```

Remarks:

On entry, bytes 1 through 4 of the parameter packet contain the 32-bit physical address (segment:address) of the screen buffer. Bytes 5 and 6 contain the length (word) of the screen buffer. The screen buffer defined must fall within the range A:0000 through B:FFFF inclusive.

On return, bytes 1 and 2 of the data packet contain the selector for the screen buffer.

This function is supported by the screen device driver.

Function 71H: Release Screen Buffer Selector

Purpose:

Function 71H releases the protected-mode selector that was acquired from a prior call to Function 70H, Get Screen Buffer Selector.

Parameter Packet Format:

```
+-------------------------------------+
| WORD  Screen buffer selector        |
+-------------------------------------+
```

where:

On entry, the parameter list contains the physical *Screen buffer selector* being released.

Data Packet Format:

```
+-------------------------------------+
| DWORD   Set to zero                 |
+-------------------------------------+
```

Remarks:

To deallocate the screen buffer selector, the screen buffer previously created by the screen device driver is passed in the parameter packet. The data packet is not used.

This function is supported by the screen device driver.

Function 72H: Get Pointer-Draw Address

Purpose:

Function 72H, which is supported by the pointer-draw device driver, is used by the mouse subsystem to obtain the entry point address of the pointer-draw routine.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
| WORD  Return code                 |
|-----------------------------------|
| DWORD Entry point address         |
|-----------------------------------|
| WORD  Data segment selector       |
+-----------------------------------+
```

where:

On return, the first Word contains a *Return code.*

The Dword contains the pointer-draw routine's *Entry point address* (FAR address, selector:offset).

The last Word contains the pointer-draw routine's *Data segment selector.*

Remarks:

Function 72H is used by the mouse subsystem to obtain the entry point of the pointer-draw routine. It is supported by the pointer-draw device driver.

The pointer-draw routine is contained within the pointer-draw device driver. It is called by the mouse device driver at interrupt time to update the pointer image on the screen. The FAR address returned by **Function 72H** is passed by the mouse subsystem to the mouse device driver through an IOCtl interface (See the mouse control IOCtl commands, **Category 07H**). The mouse device driver saves the FAR address passed and uses it when calling the pointer-draw routine at interrupt time.

Function 73H/05H: Set Video Mode

Purpose:

Function 73H, **Type 05H**, which is supported by the screen device driver, sets the current video mode.

Parameter Packet Format:

WORD	Parameter block length
WORD	Request type = 0005H

where:

Parameter block length is a word specifying the length of the parameter packet in bytes.

Request type is the type of request for this function that must be set to 0005H.

Data Packet Format:

WORD	Length of mode data
BYTE	Mode type
BYTE	Number of color bits
WORD	Text column resolution
WORD	Text row resolution
WORD	Graphics column resolution
WORD	Graphics row resolution

where:

Length of mode data is the length, in bytes of the data packet.

Mode type specifies a bitmask of video-mode characteristics, defined as follows:

Bit	Definition
0	0 = Monochrome printer adapter
	1 = Other
1	0 = Text mode
	1 = Graphics mode
2	0 = Enable color burst
	1 = Disable color burst

Number of color bits defines the number of colors as a power of 2. The possible values for the number of colors are:

Bit	Definition
1	Two colors
2	Four colors
4	Sixteen colors

Text column resolution is the number of alphanumeric columns.

Text row resolution is the number of alphanumeric rows.

Graphics column resolution is the number of pel columns.

Graphics row resolution is the number of pel rows.

Remarks:

The data packet parameters are the same as those passed to the **VioSetMode** function.

This function is supported by the screen device driver.

Function 73H/0DH: Get Video Environment

Purpose:

Function 73H, Type 0DH, which is supported by the screen device driver, sets the current video environment.

Parameter Packet Format:

```
+-------------------------------------+
| WORD   Parameter block length       |
|-------------------------------------|
| WORD   Request type = 000DH         |
|-------------------------------------|
| WORD   Size of data block           |
|-------------------------------------|
| WORD   Flags                        |
+-------------------------------------+
```

where:

Parameter block length is a word specifying the length of the parameter packet in bytes.

Request type is the type of request for this function that must be set to 000DH.

Size of data block is a word containing the length, in bytes, of the data packet to contains the video environment buffer.

Data Packet Format:

```
+---------------------------------------+
| n BYTES  Video environment buffer     |
+---------------------------------------+
```

where:

Video environment buffer is an *n*-byte structure to contain the video environment save buffer.

Remarks:

This function is supported by the screen device driver.

Function 73H/10H: Set Character Font

Purpose:

Function 73H, Type 10H, which is supported by the screen device driver, sets the current character font.

Parameter Packet Format:

```
+-------------------------------------+
| WORD   Parameter block length       |
|-------------------------------------|
| WORD   Request type = 0010H         |
|-------------------------------------|
| WORD   Number of character rows     |
+-------------------------------------+
```

where:

Parameter block length is a Word specifying the length of the parameter packet in bytes.

Request type is the type of request for this function that must be set to 0010H.

Number of character rows is a Word containing the number of character rows in the *Font information buffer*.

Data Packet Format:

```
+-------------------------------------+
| n BYTES  Font information buffer    |
+-------------------------------------+
```

where:

Font information buffer is an *n*-byte structure to contain the font information.

Remarks:

This function is supported by the screen device driver.

Function 73H/12H: Set Palette Registers

Purpose:

Function 73H, Type 12H, which is supported by the screen device driver, sets all sixteen of the palette registers.

Parameter Packet Format:

```
+-------------------------------------+
| WORD   Parameter block length       |
|-------------------------------------|
| WORD   Request type = 0012H         |
+-------------------------------------+
```

where:

Parameter block length is a Word specifying the length of the parameter packet in bytes.

Request type is the type of request for this function that must be set to 0012H.

Data Packet Format:

```
+-------------------------------------+
| WORD   Length of data               |
|-------------------------------------|
| BYTE   Palette register 0           |
|-------------------------------------|
| BYTE   Palette register 1           |
+-------------------------------------+
                :
                :
+-------------------------------------+
| BYTE   Palette register 15          |
+-------------------------------------+
```

where:

Length of data is a Word specifying the length of the data packet in bytes.

Palette register 0 through *Palette register 15* contain the settings for each of the palette registers.

Remarks:

This function is supported by the screen device driver.

Function 74H: Set Code Page Information

Purpose:

Function 74H, which is supported by the screen device driver, sets the current code page.

Parameter Packet Format:

```
+----------------------------------------+
| DWORD   Set to zero                    |
+----------------------------------------+
```

Data Packet Format:

```
+----------------------------------------+
| WORD   Length of data block            |
|----------------------------------------|
| WORD   Number of code page IDs         |
|----------------------------------------|
| WORD   First code page ID              |
+----------------------------------------+
                 :
                 :
+----------------------------------------+
| WORD   Nth code page ID                |
|----------------------------------------|
| n BYTES  Code page path specifier      |
+----------------------------------------+
```

where:

Length of data block specifies the total length, in bytes, of the data packet.

Number of code page IDs specifies the number of code page identifiers to be listed in the data packet.

First code page ID through the *Nth code page ID* are code page identifiers that are to be used for the code page file which were specified in the *config.sys* file.

Code page path specifier is an *n* byte structure specifying the path to the code page file which were specified in the *config.sys* file.

Remarks:

There is no parameter packet for this function.

This function is supported by the screen device driver.

4.4 Keyboard Control IOCtl Commands (Category 04H)

This section describes the keyboard control subfunctions within Category 04H of the Generic IOCtl commands. These subfunctions are listed as follows:

Function	Description
50H	Set Translate Table
51H	Set Input Mode
52H	Set Interim Character Flags
53H	Set Shift State
54H	Set Typamatic Rate and Delay
55H	Set Foreground Screen Group
56H	Set Session Manager Hot Key
57H	Set Pause Semaphore Handle
71H	Get Input Mode
72H	Get Interim Character Flags
73H	Get Shift State
74H	Get Character Data Record(s)
75H	Peek Character Data Record(s)
76H	Get Session Manager Hot Key
77H	Get Keyboard Type

Function 50H: Set Translate Table

Purpose:

Function 50H passes a new scancode-to-character translation table to the keyboard translate routine; the new table, which overlays the table currently in use, translates new keystrokes.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD Pointer to translation table |
+-----------------------------------+
```

where:

Pointer to translation table is a far pointer to an 842-byte translation table.

Remarks:

The default table is U.S. English. For detailed information about the contents of the *Translate table*, see Appendix A, "Translate Table Format."

Function 51H: Set Input Mode

Purpose:

Function 51H passes the current input mode to the keyboard device driver.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Input mode                 |
+-----------------------------------+
```

where:

Input mode is a one-byte field containing one of the following values:

Value	Meaning
00H	Cooked
80H	Raw

Remarks:

The keyboard device driver maintains the current input mode separately for each screen group. The caller can interrogate the mode by using **Function 71H**. The mode is also returned on every Read Character Data Record call, **Function 74H**, and Peek Character Data Record call, **Function 75H**. The default input mode is cooked. The device driver uses the mode only when it is processing CONTROL-C and CONTROL-BREAK key sequences.

Function 52H: Set Interim Character Flags

Purpose:

Function 52H sets the interim character flags maintained by the keyboard device driver.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Flags                      |
+-----------------------------------+
```

where:

Flags is a Byte field that specifies flag bits. A bit set to one indicates the following states:

Bit	Meaning
0–4	Reserved=0
5	Program requested on-the-spot conversion
6	Reserved=0
7	Interim console flag on

Remarks:

The keyboard device driver maintains the interim character flags separately for each screen group. The keyboard device driver passes the interim character flags (with each character data record) to the keyboard monitors.

Note

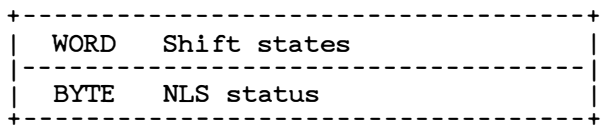
The interim character flags defined here have a different meaning from the interim character flags in a character data record returned through **Function 74H** or **75H**.

Function 53H: Set Shift State

Purpose:

Function 53H sets the current shift state for the keyboard.

Parameter Packet Format:



where:

Shift states is a Word that specifies flag bits defining universal shift states. A bit set equal to one indicates the following state:

Bit	Meaning
0	Right SHIFT key down
1	Left SHIFT key down
2	CONTROL key down
3	ALT key down
4	SCROLL LOCK on
5	NUMLOCK on
6	CAPSLOCK on
7	INSERT on
8	Left CONTROL key down
9	Left ALT key down
10	Right CONTROL key down
11	Right ALT key down
12	SCROLL LOCK key down

13 NUMLOCK key down

14 CAPSLOCK key down

15 SYSREQ key down

NLS status is a Byte that specifies the national-language-dependent shift states. This byte is zero for the United States.

Remarks:

The keyboard device driver maintains the *Shift state* separately for each screen group. Note that this call overrides the *Shift state* set by previous shift keystrokes. Also the *Shift state* set by this function code will be overridden by any subsequent shift keystrokes. The *Shift state* is inserted into the character data record built for each incoming keystroke.

Function 54H: Set Typamatic Rate and Delay

Purpose:

Function 54H sets the keyboard typamatic rate and delay to the values specified in the request.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Delay                      |
|-----------------------------------|
| BYTE   Rate                       |
+-----------------------------------+
```

where:

Delay is a Byte specifying the typamatic delay, in seconds. A value greater than the maximum value defaults to the maximum value of 0.50 seconds.

Rate is a Byte specifying the typamatic rate, in characters per second. A value greater than the maximum value defaults to the maximum value of 10.0 characters per seconds.

Remarks:

None.

Function 55H: Set Foreground Screen Group

Purpose:

Function 55H notifies the keyboard device driver that a new foreground screen group has been activated.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE  Screen group number         |
+-----------------------------------+
```

where:

Screen group number is a Byte containing the new foreground screen group. The *Screen group number* must fall within the range from 0 to 15.

Remarks:

The keyboard device driver sets the *Shift state* of the keyboard to the state that was current when the new screen group was last active. Then the keyboard device driver then begins using the keyboard input buffer (KIB) and keystroke monitor chain associated with the new screen group.

Function 56H: Set Session Manager Hot Key

Purpose:

Function 56H changes the session manager hot key for which the keyboard device driver will scan.

Parameter Packet Format:

```
+-----------------------------------+
| WORD  Hot key                     |
|-----------------------------------|
| BYTE  Scan code make              |
|-----------------------------------|
| BYTE  Scan code break             |
|-----------------------------------|
| WORD  Hot key ID                  |
+-----------------------------------+
```

where:

Hot key is a Word specifying the setting for the session manager hot key. The bits in the *Hot key* word are defined as follows:

Bit	Meaning
0	Right SHIFT key down
1	Left SHIFT key down
2	Either CONTROL key down
3	Either ALT key down
4–7	Reserved = 0
8	Left CONTROL key down
9	Left ALT key down
10	Right CONTROL key down
11	Right ALT key down
12	SCROLL LOCK key down

13 NUMLOCK key down

14 CAPSLOCK key down

15 SYSREQ key down

Scan code make is a Byte containing the scan code of the hot key make.

Scan code break is a Byte containing the scan code of the hot key break.

Hot key ID is a Word set by the caller that identifies the session manager hot key.

Note

Scan code make and *Scan code break* are mutually exclusive; either may be specified, but not both. The use of either indicates when to recognize the hot key.

Remarks:

This request is used by the session manager to set a list of keyboard hot keys for which the keyboard device driver will scan. The new hot key applies to all screen groups. Up to sixteen hot keys can be defined by the session manager for use by the keyboard device driver.

Function 56H is successful only if it is performed by the process that initially called **Function 55H**, Set Foreground Screen Group.

The combination of the shift flags in the *Hot key* Word and the scan code Bytes allow the session manager to set the hot key to a key combination such as ALT-ESC. The hot key is triggered on detection of the scan code for the hot key break. Note that a hot key can be redefined by calling this function with the same *Hot key ID*.

Function 57H: Set Pause Semaphore Handle

Purpose:

Function 57H notifies the keyboard device driver of the semaphore that it must use when it recognizes the system PAUSE key sequence.

Parameter Packet Format:

```
+-----------------------------------+
| DWORD  Handle                     |
+-----------------------------------+
```

where:

Handle is a double-word field containing the semaphore handle obtained from a **DosCreateSem** call.

Remarks:

The keyboard device driver will not pause the system unless **Function 57H** has been made previously. This command is restricted and may only be used by the process that performed the initial IOCtl call, **Function 56H** Set Session Manager Hot Key.

Function 71H: Get Input Mode

Purpose:

Function 71H obtains the input mode of the screen group of the currently active process.

Data Packet Format:

```
+-------------------------------------+
| BYTE  Input mode                    |
+-------------------------------------+
```

where:

Input mode is a one-byte field containing one of the following values:

Value	Meaning
00H	Cooked
80H	Raw

The input mode can be set with **Function 51H**, Set Output Queue Size, and is meaningful only for CONTROL-C and CONTROL-BREAK processing.

Remarks:

None.

Function 72H: Get Interim Character Flags

Purpose:

Function 72H obtains the interim character flags maintained by the keyboard device driver.

Note

The interim character flags defined here have a different meaning than do those in a character data record returned through **Function 74H** or **75H**.

Data Packet Format:

```
+-----------------------------------+
| BYTE  Interim character           |
+-----------------------------------+
```

where:

Interim character is a Byte containing flag bits. A bit set equal to one indicates the state listed below:

Bit	Meaning
0–4	Reserved = 0
5	Program requested on-the-spot conversion
6	Reserved = 0
7	Interim console flag on

Remarks:

None.

Function 73H: Get Shift State

Purpose:

Function 73H obtains the shift state of the screen group for the currently active process.

Data Packet Format:

```
+-----------------------------------+
| WORD  Shift states                |
+-----------------------------------+
```

where:

Shift states is a word field that contains flag bits defining universal shift states. A bit set equal to one indicates the following state:

Bit	Meaning
0	Right SHIFT key down
1	Left SHIFT key down
2	CONTROL key down
3	ALT key down
4	SCROLL LOCK on
5	NUMLOCK on
6	CAPSLOCK on
7	INSERT on
8	Left CONTROL key down
9	Left ALT key down
10	Right CONTROL key down
11	Right ALT key down
12	SCROLL LOCK key down
13	NUMLOCK key down

14 CAPSLOCK key down

15 SYSREQ key down

Remarks:

The *Shift state* is set by incoming keystrokes and by **Function 53H** calls.

Function 74H: Read Character Data Record(s)

Purpose:

Function 74H obtains one or more character data records from the keyboard input buffer for the screen group of the currently active process.

Parameter Packet Format:

```
+-----------------------------------+
| WORD  Transfer count              |
+-----------------------------------+
```

where:

On entry, *Transfer count* is a Word that specifies the record transfer count. The sign bit of this Word is set to request one of the following actions:

Value	Meaning
0	Wait for the requested number of keystrokes to become available. The device driver will block the requester until all requested character data records are available and have been transferred to the caller.
1	Do not wait for the requested number of keystrokes to become available. In this case, all characters currently available will be transferred, up to the requested transfer count. A maximum of sixteen characters will be transferred.

On return, the *Transfer count* field contains the actual number of character data records transferred. The sign bit is set to either 0 if the current input mode is cooked, or 1 if the input mode is raw.

Data Packet Format:

```
+-----------------------------------+
| n CharData records                |
+-----------------------------------+
```

where:

On return, the data buffer contains one or more character data records *(CharData)*. For information about the *CharData* record structure, see the **KbdCharIn** call in the *Microsoft Operating System/2 Programmer's Reference* or Section 2.3.1, "Console Device Drivers" in this manual.

Remarks:

None.

Function 75H: Peek Character Data Record(s)

Purpose:

Function 75H obtains one character data record from the head of the keyboard input buffer of the screen group for the currently active process. The character data record is not removed from the KIB.

Parameter Packet Format:

```
+-----------------------------------+
| WORD  Keystroke status            |
+-----------------------------------+
```

where:

Keystroke status is a Word that specifies one of the following values:

Value	Meaning
0	No keystroke is available
1	A character data record is being returned

The sign bit is set to either 0 if the current input mode is cooked, or 1 if the input mode is raw.

Data Packet Format:

```
+-----------------------------------+
| n CharData Records                |
+-----------------------------------+
```

where:

On return, the data buffer contains one or more character data records *(CharData)*. For more information about the *CharData* record structure, see the **KbdCharIn** call in the *Microsoft Operating System/2 Programmer's Reference* or Section 2.3.1 "Console Device Drivers," in this manual.

Remarks:

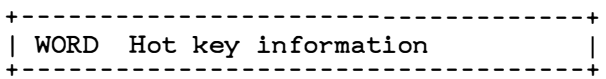
None.

Function 76H: Get Session Manager Hot Key

Purpose:

Function 76H returns the scan code that the keyboard device driver is using as the session manager hot key.

Parameter Packet Format:

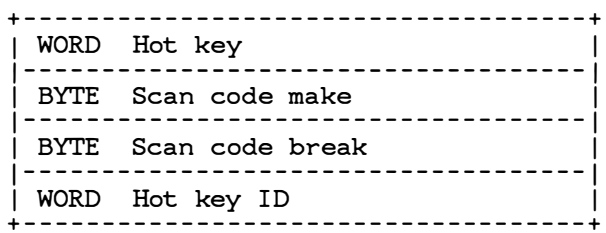


where:

Hot key information is a Word that specifies the type of information to return. On entry, this Word is defined as follows:

Value	Meaning
0	Return the maximum number of hot keys that the keyboard device driver can support.
1	Return the number of hot keys currently defined in the system and return the key information for each in the data packet.

Data Packet Format:



where:

Hot key is a word field containing the setting for the session manager hot key. The bits in the *Hot key* word are defined as follows:

Bit	Meaning
0	Right SHIFT key down
1	Left SHIFT key down
2	Either CONTROL key down
3	Either ALT key down
4–7	Reserved = 0
8	Left CONTROL key down
9	Left ALT key down
10	Right CONTROL key down
11	Right ALT key down
12	SCROLLLOCK key down
13	NUMLOCK key down
14	CAPSLOCK key down
15	SYSREQ key down

Scan code make is a Byte containing the scan code of the hot key make.

Scan code break is a Byte containing the scan code of the hot key break.

Note

Scan code make and *Scan code break* are mutually exclusive; either may be specified, but not both. The use of either indicates when the the hot key is recognized.

Hot key ID is a Word set by the caller that identifies the session manager hot key.

Remarks:

If the Word in the parameter packet was one on entry, then one or more hot key data structures (as defined in the data packet) and returned.

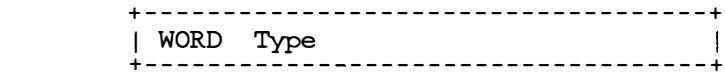
Function 76H first should be called with the *Hot key information* Word equal to zero to determine the maximum number of hot keys that can be supported by the device driver. The value returned should be used to determine the required size of the data buffer on a subsequent call (with *Hot key information* equal to one) to return the hot key data structures.

Function 77H: Get Keyboard Type

Purpose:

Function 77H returns information about the type of keyboard being used.

Data Packet Format:



where:

Type is a Word that specifies the keyboard type being used. The recognized values for the *Type* Word are defined as follows:

Value	Meaning
00H	PC/AT keyboard
01H	Enhanced keyboard
02–0FFH	Reserved

Remarks:

None.

4.5 Printer Control IOCTL Commands (Category 05H)

This section describes the printer control subfunctions within Category 05H of the Generic IOCtl commands. These subfunctions are listed as follows:

Function	Description
42H	Set Frame Control
44H	Set Infinite Retry
46H	Initialize Printer
62H	Get Frame Control
64H	Get Infinite Retry
65H	Get Printer Status

Function 42H: Set Frame Control

Purpose:

Function 42H sets frame control for a print device.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE    Command information      |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| BYTE   Characters per line        |
|-----------------------------------|
| BYTE   Lines per inch             |
+-----------------------------------+
```

where:

Characters per line is a Byte containing the number of characters allowed on a line, either 80 or 132.

Lines per inch is a Byte containing the allowed number of lines per inch, either 6 or 8.

Function 44H: Set Infinite Retry

Purpose:

Function 44H sets infinite retry for a print device.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE  Command information        |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| BYTE   Retry                      |
+-----------------------------------+
```

where:

Retry is a Byte defined as follows:

Value	Meaning
0	Disable infinite retry
1	Enable infinite retry

Function 46H: Initialize Printer

Purpose:

Function 46H initializes a print device.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE   Command information       |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

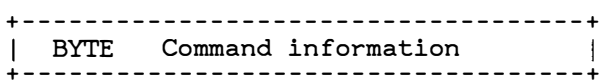
The data packet must be set to zero.

Function 62H: Return Frame Control

Purpose:

Function 62H returns frame control information for a print device.

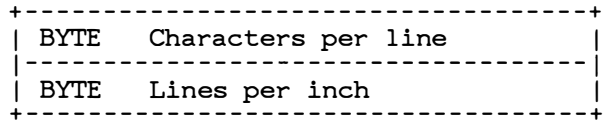
Parameter Packet Format:



where:

Command information is a reserved Byte that must be zero.

Data Packet Format:



where:

Characters per line is a one-byte field containing the number of characters allowed on a line, either 80 or 132.

Lines per inch is a one-byte field containing the allowed number of lines per inch, either 6 or 8.

Function 64H: Return Infinite Retry

Purpose:

Function 64H causes an infinite retry for a print device.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE    Command information      |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
|  BYTE   Retry                     |
+-----------------------------------+
```

where:

On return, the *Retry* field contains one of the following:

Value	Meaning
0	Infinite retry is disabled.
1	Infinite retry is enabled.

Function 65H: Return Printer Status

Purpose:

Function 65H returns the status of a print device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| BYTE   Error                      |
+-----------------------------------+
```

where:

The *Error* byte is set as follows:

Bit	Meaning
0	Set if timeout
1	Reserved
2	Reserved
3	Set if I/O error
4	Set if selected
5	Set if out of paper
6	Set if acknowledge
7	Set if not busy

4.6 Mouse Control IOCtl Commands (Category 07H)

This section describes the mouse control subfunctions within Category 07H of the Generic IOCtl commands. These subfunctions are listed as follows:

Function	Description
50H	Allow Pointer Drawing after Screen Switch
51H	Update Screen Display Mode
52H	Screen Switcher Notification
53H	Set New Scaling Factors
54H	Set New Event Mask
55H	Set System Hot Key Button
56H	Set Pointer Shape
57H	Unmark Collision Area
58H	Mark Collision Area
59H	Set Pointer Screen Position
5AH	Set Protected-Mode Pointer-Draw Address
5BH	Set Real-Mode Pointer-Draw Address
5CH	Set Mouse Status Flags
60H	Get Number of Buttons
61H	Get Number of Mickeys/centimeter
62H	Get Device Status Flags
63H	Read Event Queue
64H	Get Event Queue Status
65H	Get Current Event Mask
66H	Get Current Scaling Factors
67H	Get Pointer Screen Position

68H	Get Pointer Shape
69H	Get System Hot Key Button

Function 50H: Allow Pointer Drawing after Screen Switch

Purpose:

Function 50H is used by the screen switcher to notify the mouse device driver that a screen group switch has been completed and the pointer may now be drawn.

Parameter Packet Format:

```
+-----------------------------------+
|  DWORD  Set to zero               |
+-----------------------------------+
```

Data Packet Format:

```
+-----------------------------------+
|  DWORD  Set to zero               |
+-----------------------------------+
```

Remarks:

This command has no input or output parameters.

Function 51H: Update Screen Display Mode

Purpose:

Function 51H notifies the mouse device driver that the screen display mode has been modified.

Parameter Packet Format:

The parameter packet points to a *Mode data definition* record with the following format:

```
+-----------------------------------+
| WORD    Length of data structure  |
|-----------------------------------|
| BYTE    Type                      |
|-----------------------------------|
| BYTE    Color                     |
|-----------------------------------|
| WORD    Text column resolution    |
|-----------------------------------|
| WORD    Text row resolution       |
|-----------------------------------|
| WORD    Graphics column resolution|
|-----------------------------------|
| WORD    Graphics row resolution   |
+-----------------------------------+
```

Length of data structure specifies the length of the *Mode data definition record* (that is, the length of the data packet) in bytes.

The *Type* specifies a bitmask of mode characteristics, defined as follows:

Bit	Definition
0	0 = Monochrome printer adapter
	1 = Other
1	0 = Text mode
	1 = Graphics mode

2	0 = Enable color burst 1 = Disable color burst

Color defines the number of colors as a power of 2. The possible values for the number of colors are:

Bit	Definition
1	Two colors
2	Four colors
4	Sixteen colors

Text column resolution is the number of alphanumeric columns.

Text row resolution is the number of alphanumeric rows.

Graphics column resolution is the number of pel columns.

Graphics row resolution is the number of pel rows.

Remarks:

This function does not provide return parameter values.

For more information about the parameters in the set mode operation, see the **VioSetMode** system call in the *Microsoft Operating System/2 Programmer's Reference.*

Note

Whenever the VIO subsystem or registered VIO subsystem sets or resets the display mode, it must synchronize the mouse device driver's pointer-update routines by providing this notification record to the mouse driver prior to switching display modes.

Function 52H: Screen Switcher Notification

Purpose:

Function 52H is used by the system screen switcher to notify the mouse device driver that a screen group switch is about to take place. It also sets a system pointer-draw enable/disable flag, effectively locking out any pointer drawing until the flag is cleared by Function 50H, Allow Pointer Drawing after Screen Switch.

Parameter Packet Format:

```
+-----------------------------------+
|  WORD   Screen group number       |
|-----------------------------------|
|  WORD   Notification type         |
+-----------------------------------+
```

where:

On entry, the *Screen group number* is a word containing the screen group number for notification action. The value of this number must be in the range:

$0 \leq$ *screen group number* $\leq$ *MaxNumberOfScreenGroups*

The GDT segment defines the valid range for screen group identifiers.

Notification type is a word containing the notification type of the switch. The values for this parameter are:

Value	Meaning
–1	The specified *Screen group number* is terminating.
≥ 0	The specified *Screen group number* is being switched to.

Remarks:

None.

Function 53H: Set New Scaling Factors

Purpose:

Function 53H reassigns the scaling factors of the current pointing device.

Parameter Packet Format:

```
+-----------------------------------+
|   WORD   X scaling factor         |
|-----------------------------------|
|   WORD   Y scaling factor         |
+-----------------------------------+
```

where:

On entry, *X scaling factor* is the new x (row) coordinate scaling factor.

Y scaling factor is the new y (column) coordinate scaling factor. Scaling factors must be positive integer values in the range:

$0 < value \leq (32K - 1)$

Remarks:

Function 53H does not provide return parameter values.

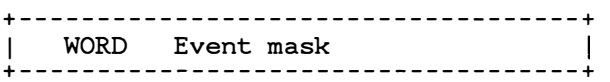
Scaling factors are ratio values that determine how much relative movement is necessary before the mouse device driver will report a mouse event. In graphics mode, the ratio is mickeys per pixel. In text mode, the ratio is mickeys per character. The ratio values default to one mickey per row unit and one mickey per column unit.

Function 54H: Set New Event Mask

Purpose:

Function 54H reassigns a new pointing-device event mask.

Parameter Packet Format:



where:

Event mask has the following bit-level definitions:

Bit	Meaning
0	Set if all mouse motion, no buttons pressed
1	Set if motion with button 1 is pressed
2	Set if button 1 is pressed
3	Set if motion with button 2 is pressed
4	Set if button 2 is pressed
5	Set if motion with button 3 is pressed
6	Set if button 3 is pressed
7–15	Reserved = 0

Remarks:

None.

Function 55H: Set System Hot Key Button

Purpose:

Function 55H sets the mouse button equivalent for the system hot key.

Parameter Packet Format:

```
+-----------------------------------+
|   WORD   Button bitmap            |
+-----------------------------------+
```

where:

On entry, the parameter list requires a Word that specifies the *Button bitmap* indicating the desired mouse button equivalent for the system hot key. This parameter is bitmapped as follows:

Bit	Meaning
0	Set if no system hot key desired
1	Set if button 1 = system hot key
2	Set if button 2 = system hot key
3	Set if button 3 = system hot key
4–15	Reserved = 0

Remarks:

If bit 0 is set, no system hot key support is provided, regardless of the other bit settings in the Word value. If multiple bits are set, not including bit 0, the system hot key is interpreted as requiring the indicated buttons to be pressed simultaneously.

Function 55H may only be altered by the process that initially issues it. It should be used only by the command shell.

Function 56H: Set Pointer Shape

Purpose:

Function 56H sets the shape of the pointer.

Parameter Packet Format:

```
+-----------------------------------+
|  WORD   Pointer-image height      |
|-----------------------------------|
|  WORD   Pointer-image width       |
|-----------------------------------|
|  WORD   Row offset to hot spot    |
|-----------------------------------|
|  WORD   Column offset to hot spot |
|-----------------------------------|
|  WORD   Length of ptr-image buffer|
+-----------------------------------+
```

where:

Pointer-image height is a Word that contains the height value (in rows) of the pointer image.

Pointer-image width is a Word that contains the width value (in columns) of the pointer image.

Row offset is a Word that contains the row offset within the pointer image to the hot spot.

Column offset is a Word that contains the column offset within the pointer image to the hot spot.

Length of pointer-image buffer is the length of the pointer shape's buffer in bytes; this should be equal to height-value times width-value.

Data Packet Format:

```
+-----------------------------------+
|  n BYTE Pointer-image buffer      |
+-----------------------------------+
```

where:

The data packet contains the *n*-byte *Pointer-image buffer*. The format of the pointer image depends on the mode of the display. For currently supported modes, the buffer always consists of the AND pointer-image data followed by the XOR pointer-image data, and the buffer always describes only one display plane.

Remarks:

All the pointer definition record fields and the pointer-image buffer are validated using the screen group's mode table values. The parameter values must be the same mode as the current screen group's display mode. For text mode, these values must be character values; for graphics mode, they must be pixel values.

Function 57H: Unmark Collision Area

Purpose:

Function 57H unmarks the current collision area, allowing the pointer to be drawn anywhere on the screen.

Parameter Packet Format:

```
+------------------------------+
|  DWORD Set to zero           |
+------------------------------+
```

Data Packet Format:

```
+------------------------------+
|  DWORD Set to zero           |
+------------------------------+
```

Remarks:

There are no input or output parameters for this call.

If a collision area has been declared for the screen group, that area is released and the pointer position is checked. If the pointer was in the released area, then it is drawn, but if the pointer was not in the released area, the pointer-draw operation takes place.

Function 58H: Mark Collision Area

Purpose:

Function 58H reserves a new collision area for use by the device driver.

Parameter Packet Format:

```
+-----------------------------------+
|  WORD  Pointer to collision buffer |
+-----------------------------------+
```

where:

Pointer to collision buffer is an address pointing to a four-word structured buffer. On entry, this buffer defines the collision area that will be protected from being overwritten by system pointer-draw operations. The collision area buffer contains the following field definitions:

Length	Description
WORD	Starting *x*-coordinate, *X_position*
WORD	Starting *y*-coordinate, *Y_position*
WORD	Height of collision area, *X_height*
WORD	Width of collision area, *Y_width*

These fields may be specified in either character or pixel values as long as the orientation is preserved across all values in the buffer record.

Remarks:

The pointer is not drawn in the collision area until a different area is specified with another call of this command.

If the collision area is defined as the entire screen, pointer-draw operations are effectively disabled for the entire screen group.

Function 59H: Set Pointer Screen Position

Purpose:

Function 59H assigns a new screen position for the pointer image.

Parameter Packet Format:

```
+--------------------------------------+
|  WORD  X-coordinate of pointer image |
|--------------------------------------|
|  WORD  Y-coordinate of pointer image |
+--------------------------------------+
```

where:

On entry, the *x*-coordinate is a Word that specifies the new *x* (row) coordinate of the pointer.

On entry, the *y*-coordinate is a Word that specifies the pointer *y* (column) coordinate of the pointer.

Remarks:

The coordinate values depend on the display mode. Character position values must be used if the display is in text mode. Pixel values must be used if the display is in graphics mode.

This call has no effect on current collision area definitions. If a pointer image is already defined for the screen group, it is replaced by the new pointer image.

If the pointer image is directed into an existing collision area, it will remain hidden (invisible) until either enough mouse movement is generated to place it outside of the collision area, or until the collision area is released.

Function 5AH: Set Protected-mode Pointer-draw Address

Purpose:

Function 5AH notifies the mouse device driver of the address of a protected-mode screen group's pointer-draw routine.

Parameter Packet Format:

```
+-----------------------------------+
|  WORD   Selector                  |
|-----------------------------------|
|  WORD   Offset                    |
+-----------------------------------+
```

where:

On entry, the *Selector* and *Offset* fields make up the selector:offset pair of the address for the pointer-draw device driver's entry point.

Remarks:

Function 5AH is valid for protected-mode only.

The pointer-draw routine is an installed, pseudo-character device driver. The mouse handler must

- Open the pointer-draw device driver.
- Query the pointer-draw device driver for the address of its entry point.
- Pass the resulting address of the pointer-draw entry point to the mouse device driver using this function.

The mouse device driver issues a FAR call to the pointer-draw device driver whenever a mouse interrupt occurs that requires action concerning the pointer image. In addition, the mouse device driver may call the pointer-draw routine as a result of some action on the part of the application, such as

MouDrawPtr
MouRemovePtr
MouSetPtrPos
MouSetPtrShape
MouGetPtrShape

Function 5BH: Set Real-Mode Pointer-Draw Address

Purpose:

Function 5BH notifies the real-mode mouse device driver of the entry point of a real-mode screen group's pointer-draw routine. It is issued only by the shell (session manager) at the end of system initialization.

Parameter Packet Format:

```
+-----------------------------------+
|  WORD   Selector                  |
|-----------------------------------|
|  WORD   Offset                    |
+-----------------------------------+
```

where:

On entry, the *Selector*, and the *Offset* that make up the selector:offset pair of the address for the pointer-draw device driver's entry point.

Remarks:

Function 5BH is valid for real-mode only.

Function 5CH: Set Mouse Status Flags

Purpose:

Function 5CH sets a subset of the current mouse device driver status flags.

Parameter Packet Format:

```
+-----------------------------------+
| WORD   Status flags               |
+-----------------------------------+
```

where:

On entry, *Status flags* is a Word that specifies the status flags for an input device. This field is defined with the following bit-level definitions:

Bit	Meaning
0	Set if event queue busy with I/O
1	Set if block read in progress
2–7	Reserved = 0
8	Set if the interrupt-level pointer-draw routine is not called
9	Set if mouse data is being returned in mickeys (not pixels)
10–15	Reserved = 0

Remarks:

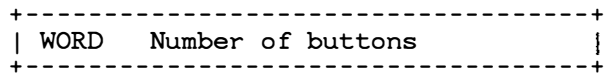
None.

Function 60H: Get Number of Buttons

Purpose:

Function 60H returns the current pointing device configuration indicating the number of buttons supported by the mouse device driver.

Data Packet Format:



where:

On return, the application's data buffer contains a Word that specifies the *Number of buttons*. This value may be one of the following:

Value	Meaning
1	One button support
2	Two button support
3	Three button support

Remarks:

Function 60H does not require input parameters.

Function 61H: Get Number of Mickeys

Purpose:

Function 61H returns the current pointing device configuration for the number of mickeys per centimeter.

Data Packet Format:

```
+-----------------------------------+
|  WORD  Number of mickeys/cm       |
+-----------------------------------+
```

where:

On return, the application storage area contains a Word (written by the mouse device driver) that specifies the current number of mickeys per centimeter. The *Number of mickeys/cm* is a positive integer in the range:

$0 < mickeys/centimeter \leq (32K - 1)$

Remarks:

Function 61H does not require input parameters.

Function 62H: Get Device Status Flags

Purpose:

Function 62H returns the current status flags of the pointing device driver.

Data Packet Format:

```
+-----------------------------------+
|  WORD  Status flags               |
+-----------------------------------+
```

where:

On return, the data packet contains a Word that specifies *Status flags*. The return value is defined as follows:

Bit	Meaning
0	Set if event queue busy with I/O
1	Set if block read in progress
2–7	Reserved = 0
8	Set if the interrupt-level pointer-draw routine is not called
9	Set if mouse data is being returned in mickeys (not pixels)
10–15	Reserved = 0

Remarks:

Function 62H does not require input parameters.

Function 63H: Read Event Queue

Purpose:

Function 63H reads the event queue for the pointing device.

Parameter Packet Format:

```
+-----------------------------------+
|   WORD  Read type                 |
+-----------------------------------+
```

where:

On entry, the *Read type* parameter determines the type of action to be taken only if no event-queue data is available. The value of *Read type* may be one of the following:

Value	Meaning
0	Return a null record (No Wait).
1	Block the process (Wait) until event data is available for the request.

Data Packet Format:

```
+-----------------------------------+
|  10 BYTES  Event buffer           |
+-----------------------------------+
```

where:

On return, the data packet contains the event queue's FIFO 10-byte *Event buffer* record element, which has the following format:

```
EventMask    WORD     ;see Function 65H
EventTime    DWORD    ;event time stamp
                      ;(standard time-of-day format)
RowPos       WORD     ;pointer row coordinate
ColPos       WORD     ;pointer column coordinate
```

Function 64H: Get Event Queue Status

Purpose:

Function 64H returns both the current number of queued elements in the event queue, and the maximum number of elements allowed in an event queue.

Data Packet Format:

```
+-----------------------------------+
| WORD  Number of queue elements    |
|-----------------------------------|
| WORD  Max. num. queue elements    |
+-----------------------------------+
```

where:

On return, the data packet contains two one-word parameters. The first parameter, *NumberOfQueueElements*, contains the current number of event queue elements. This return value is in the range of:

$0 \leq value \leq MaxNumQueueElements$

The *MaxNumQueueElements* field contains a Word that specifies the maximum number of queue elements (configured by the mouse device driver).

Remarks:

Function 64H does not require input parameters.

Function 65H: Get Current Event Mask

Purpose:

Function 65H returns the event mask of the current pointing device.

Data Packet Format:

```
+-----------------------------------+
|   DWORD  Event mask               |
+-----------------------------------+
```

where:

On return, the data packet contains a Word that specifies the *Event mask*. This value could be any valid combination of enabled or disabled event flags. The *Event mask* has the following bit-level definitions:

Bit	Meaning
0	Set if all mouse motion, no buttons pressed
1	Set if motion with button 1 is pressed
2	Set if button 1 is pressed
3	Set if motion with button 2 is pressed
4	Set if button 2 is pressed
5	Set if motion with button 3 is pressed
6	Set if button 3 is pressed
7–15	Reserved = 0

Remarks:

Function 65H does not require input parameters.

Function 66H: Get Current Scaling Factors

Purpose:

Function 66H returns the scaling factors of the current pointing device.

Data Packet Format:

```
+-----------------------------------+
|  WORD  X-coordinate scaling factor |
|-----------------------------------|
|  WORD  Y-coordinate scaling factor |
+-----------------------------------+
```

where:

On return, the data packet contains two one-word parameters, written by the mouse device driver. The first parameter contains the scaling factor of the current x (row) coordinate. The second parameter contains the scaling factor of the current y (column) coordinate.

The *Scaling factor* values are positive integers in the range:

$0 < value \leq (32K - 1)$

Remarks:

Function 66H does not require input parameters.

Scaling factors are the ratio values that determine how much relative movement is necessary before the mouse device driver reports a mouse event. In graphics mode, this ratio is in mickeys per pixel. In text mode, this ratio is in mickeys per character. The ratio values default to one mickey per row unit and one mickey per column unit.

Function 67H: Get Pointer Screen Position

Purpose:

Function 67H returns the position of the current screen's pointer image.

Data Packet Format:

```
+-------------------------------------------+
|WORD   X-coordinate of pointer image       |
|-------------------------------------------|
|WORD   Y-coordinate of pointer image       |
+-------------------------------------------+
```

where:

On return, the data packet contains two one-word parameters containing the pointer-image coordinates. The first parameter is a Word that specifies the pointer's new x (row) coordinate. The second parameter is a Word that specifies the pointer's y (column) coordinate.

Remarks:

Function 67H does not require input parameters.

The coordinate values depend on the display mode. If the display is in text mode, character position values must be used. But if the display is in graphics mode, pixel values must be used.

Function 68H: Get Pointer Shape

Purpose:

Function 68H returns the current mouse pointer image.

Parameter Packet Format:

WORD	Pointer-image height
WORD	Pointer-image width
WORD	Row offset to hot spot
WORD	Column offset to hot spot
DWORD	Length of pointer-image buffer

where:

On return, the parameter list contains five parameters:

Pointer-image height is a word that contains the height value (in rows) of the pointer image.

Pointer-image width is a Word that specifies the width (in columns) of the pointer image.

Row offset contains the row offset within the pointer image to the hot spot.

Column offset contains the column offset within the pointer image to the hot spot.

Length of pointer-image buffer is the length of the pointer shape's buffer, in bytes; this value should be equal to height-value times width-value.

Data Packet Format:

```
+-----------------------------------+
|  BYTE Pointer-image buffer        |
+-----------------------------------+
```

where:

On return, the data packet contains the current *Pointer-image buffer*. The format of the pointer image depends on the mode of the display. For currently supported modes, the buffer always consists of the AND pointer-image data followed by the XOR pointer-image data.

Returns:

This function will exit in a normal state if the input pointer-image buffer length is greater than or equal to the amount of storage required for the pointer-image. Also, the current pointer information will be returned in the pointer data record, and the pointer-image data will be copied into the data packet buffer.

An "invalid buffer size" error will occur if the length of the input pointer-image buffer is smaller than the amount of storage necessary to perform the data copy. Also, the buffer length that is returned will be the minimum value.

Remarks:

The parameter values are in the same mode as the current screen group's display mode. For text mode, these values are character values; for graphics mode, they are pixel values.

On input, the only field in the pointer definition record that is used by the mouse device driver is the *Length of pointer-image buffer*. This buffer is pointed to by the data packet parameter.

Function 69H: Get System Hot Key Button

Purpose:

Function 69H returns the mouse button equivalent for the system hot key.

Data Packet Format:

```
+-----------------------------------+
|  WORD   Hot key button mask       |
+-----------------------------------+
```

where:

On return, *Hot key button mask* is a Word that specifies the button bitmap indicating the desired mouse button equivalent for the system hot key. This parameter is defined as follows:

Bit	Meaning
0	Set if no system hot key is desired
1	Set if button 1 = system hot key
2	Set if button 2 = system hot key
3	Set if button 3 = system hot key
4–15	Reserved = 0

Remarks:

Function 69H does not require input parameters.

If bit 0 is set, no system hot key support is provided regardless of the other bit settings in the Word value. If multiple bits are set, not including bit 0, the system hot key is interpreted as requiring the indicated buttons to be pressed simultaneously.

4.7 Disk/Diskette Control IOCtl Commands (Category 08H)

This section describes the disk/diskette control subfunctions within Category 08H of the Generic IOCtl commands. These subfunctions are listed as follows:

Function	Description
00H	Lock Drive
01H	Unlock Drive
02H	Redetermine Media
03H	Set Logical Map
20H	Block Removable
21H	Get Logical Map
43H	Set Device Parameters
44H	Write Track
45H	Format and Verify Track on Drive
63H	Get Device Parameters
64H	Read Track
65H	Verify Track

Function 00H: Lock Drive

Purpose:

Function 00H locks a drive.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Command information        |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

The operation of locking a drive is used to exclude file I/O by another process on the volume in the drive. **Function 00H** succeeds only if there is one file handle open on the volume in the drive. This is necessary since the desired result is to exclude all other I/O to the volume.

Function 01H: Unlock Drive

Purpose:

Function 01H unlocks a drive.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Command information        |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

The locked volume represented by the handle is required in the drive.

Function 02H: Redetermine Media

Purpose:

Function 02H redetermines media on a block device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Command information        |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

This function updates the volume in the drive. It is normally issued after the volume ID information on the volume has been changed (such as by formatting the disk). **Function 02H** should be done only to a locked volume.

Function 03H: Set Logical Map

Purpose:

Function 03H sets the logical drive mapping for a block device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE    Command information       |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| BYTE   Logical drive number       |
+-----------------------------------+
```

where:

On entry, the *Logical drive number* field contains the logical drive number (1=A, 2=B, ...). On return, this field contains the logical drive currently mapped to the drive that the specified file handle is open on. If only one logical device is mapped onto the physical drive, this byte is set to zero.

Remarks:

None.

Function 20H: Block Removable

Purpose:

Function 20H marks whether the block device is removable.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE  Command information         |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| BYTE  Media type                  |
+-----------------------------------+
```

where:

On return, the *Media type* field is set accordingly:

Value	Meaning
0	Removable media
1	Nonremovable media

Remarks:

None.

Function 21H: Get Logical Map

Purpose:

Function 21H returns the mapping of a logical drive.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE   Command information        |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| BYTE   Logical drive number       |
+-----------------------------------+
```

where:

On entry, the *Logical drive number* is a field that contains the logical drive number (1=A, 2=B, ...). On return, this field contains the logical drive that is currently mapped to the drive that the specified handle was opened on. This byte is set to zero if only one logical drive is mapped onto this physical drive.

Remarks:

None.

Function 43H: Set Device Parameters

Purpose:

Function 43H sets device parameters for MS OS/2 block devices.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command Information          |
+-----------------------------------+
```

where:

The *Command information* field is a bitfield defined as follows:

Bit	Value	Description
0	0	Build the BIOS Parameter Block (BPB) from the medium for all subsequent Build BPB requests.
	1	Change the default BPB for the physical device.
1	0	Change the BPB for the medium to the specified BPB, and return the new BPB as the BPB for the medium for all subsequent Build BPB requests.
2–7	0	Reserved

Data Packet Format:

31 BYTES Extended BPB structure
WORD Number of cylinders
BYTE Device type
WORD Device attributes

where:

The *Extended BPB structure* has the following format:

WORD	Bytes per sector
BYTE	Sectors per cluster
WORD	Reserved sectors
BYTE	Number of FATs
WORD	Root directory entries
WORD	Total sectors
BYTE	Media descriptor
WORD	Sectors per FAT
WORD	Sectors per track
WORD	Number of heads
DWORD	Hidden sectors
DWORD	Large total sectors

The *Number of cylinders* field indicates the number of cylinders defined for the physical device.

The *Device type* field, which describes the physical layout of the specified device, takes one of the following values:

Value	Meaning
0	48 tracks per inch, low-density floppy drive
1	96 tracks per inch, high-density floppy drive
2	3.5 inch (720K) floppy drive
3	8-inch, single-density floppy drive
4	8-inch, double-density floppy drive
5	Hard disk
6	Tape drive
7	Other (unknown type of device)

The *Device attributes* field is a bitfield that returns various flag information about the specified drive:

Bit	Meaning
0	Removable media flag. If set, the media can be changed.
1	Changeline flag. If set, the media can detect when it has been changed.
2–15	Reserved = 0

Remarks:

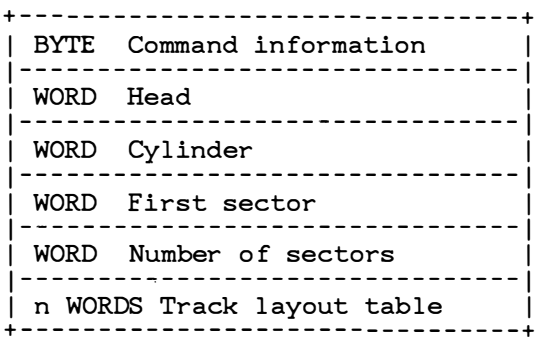
None.

Function 44H: Write Track

Purpose:

Function 44H writes to a track on a drive.

Parameter Packet Format:



where:

The *Command information* field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1, and contains only consecutive sectors.
2–7	Reserved = 0

The *Head* field contains the physical head on the drive to perform the write operation.

The *Cylinder* field contains the cylinder number for the read/write/verify.

The *First sector* field contains the logical sector number within the *Track layout table* to start the I/O operation.

Note

The *Track layout table* is based from zero (Origin 0), so the third sector is numbered 2.

The *Number of sectors* field contains the number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions will *not* step heads/tracks).

The *Track layout table* field is as follows:

```
+-----------------------------------+
| WORD  Sector number for sector 1  |
|-----------------------------------|
| WORD  Sector size for sector 1    |
|-----------------------------------|
| WORD  Sector number for sector 2  |
|-----------------------------------|
| WORD  Sector size for sector 2    |
|-----------------------------------|
| WORD  Sector number for sector 3  |
|-----------------------------------|
| WORD  Sector size for sector 3    |
+-----------------------------------+
                :
                :
+-----------------------------------+
| WORD  Sector number for sector n  |
|-----------------------------------|
| WORD  Sector size for sector n    |
+-----------------------------------+
```

The specified sector table provides information that is used during write operations.

Data Packet Format:

```
+-----------------------------------+
| BYTES Buffer                      |
+-----------------------------------+
```

where:

The data packet is a *Buffer*. For the Write function, it contains the data to be written.

Remarks:

This function performs the write operation to the device specified in this request. The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into *n* single-sector operations, and one sector at a time is written.

Function 45H: Format and Verify Track on Drive

Purpose:

Function 45H formats and verifies a track on a drive.

Parameter Packet Format:

```
+---------------------------------+
|  BYTE  Command information      |
|---------------------------------|
|  WORD  Head                     |
|---------------------------------|
|  WORD  Cylinder                 |
|---------------------------------|
|  WORD  Reserved                 |
|---------------------------------|
|  WORD  Number of sectors        |
|---------------------------------|
|  BYTES Format track table       |
+---------------------------------+
```

where:

The *Command information* field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1 and contains only consecutive sectors.
2–7	Reserved = 0.

The *Head* field contains the number of the physical head on which to perform the operation.

The *Cylinder* field contains the cylinder number for the operation.

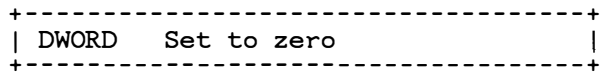
The *Number of sectors* field contains the number of sectors on the track being formatted.

The *Format track table* field contains four byte-tuples. Each 4-tuple is in the form (*c*, *h*, *r*, *n*) with *c* = cylinder number, *h* = head number, *r* = sector ID, and *n* = bytes per sector.

n	**Bytes/sector**
0	128
1	256
2	512
3	1024

There is a 4-tuple for each sector in the track to be formatted. All the cylinder and head numbers must be the same.

Data Packet Format:



Remarks:

Function 45H formats and verifies the track specified according to the information passed in the *Track layout* field. The track layout is passed to the controller and the controller performs whatever operations are necessary to do the formatting.

Note that some controllers will *not* support formatting tracks with varying sector sizes, so in general the application writer must make sure that the sector sizes specified in the *Track layout* table are all the same.

Function 63H: Get Device Parameters

Purpose:

Function 63H returns device parameters for an MS OS/2 block device.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

The *Command information* field is a bitfield defined as follows:

Bit	Value	Description
0	0	Return the recommended BPB for the drive. The recommended BPB for the drive is the BPB for the physical device.
	1	Return the BPB for the media currently in the drive.
1–7	0	Reserved

Data Packet Format:

```
+-----------------------------------+
|  13 BYTES  BPB for device         |
|-----------------------------------|
|  WORD   Number of cylinders       |
|-----------------------------------|
|  BYTE   Device type               |
|-----------------------------------|
|  WORD   Device attributes         |
+-----------------------------------+
```

where:

The device driver maintains two BIOS Parameter Blocks (BPBs) for each drive: one is the current BPB (that corresponds to the media), and the other is a recommended BPB based on the type of media corresponding to the physical device (for a high-density drive, it is a BPB for a 96tpi (tracks per inch) floppy; for a low-density drive, it is the BPB for a 48tpi floppy, etc). The low bit of the *Command information* field (in the Parameter Packet) indicates which BPB the application would like to see.

The *Number of cylinders* field indicates the number of cylinders defined.

The *Device type* field, which describes the physical layout of the specified device, takes one of the following values:

Value	Meaning
0	48 tracks per inch, low-density floppy drive
1	96 tracks per inch, high-density floppy drive
2	3.5 inch (720K) floppy drive
3	8-inch, single-density floppy drive
4	8-inch, double-density floppy drive
5	Hard disk
6	Tape drive
7	Other (unknown type of device)

The *Device attributes* field is a bitfield that returns various flag information about the specified drive:

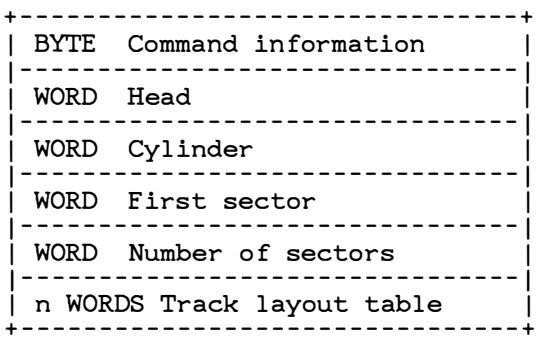
Bit	Description
0	Removable media flag. If set, the media can be changed.
1	Changeline flag. If set, the media can detect when it has been changed.
2–15	Reserved = 0

Function 64H: Read Track

Purpose:

Functions 64H reads from a track on a drive.

Parameter Packet Format:



where:

The *Command information* field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1, and contains only consecutive sectors.
2–7	Reserved = 0

The *Head* field contains the physical head on the drive to perform the read operation.

The *Cylinder* field contains the cylinder number for the read.

The *First sector* field contains the logical sector number within the *Track layout table* to start the I/O operation.

Note

The *Track layout table* is based from zero (Origin 0), so the third sector is numbered 2.

The *Number of sectors* field contains the number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions will *not* step heads/tracks).

The *Track layout table* field is as follows:

```
+-----------------------------------+
| WORD  Sector number for sector 1  |
|-----------------------------------|
| WORD  Sector size for sector 1    |
|-----------------------------------|
| WORD  Sector number for sector 2  |
|-----------------------------------|
| WORD  Sector size for sector 2    |
|-----------------------------------|
| WORD  Sector number for sector 3  |
|-----------------------------------|
| WORD  Sector size for sector 3    |
+-----------------------------------+
                :
                :
+-----------------------------------+
| WORD  Sector number for sector n  |
|-----------------------------------|
| WORD  Sector size for sector n    |
+-----------------------------------+
```

The specified sector table provides information that is used during read operations.

Data Packet Format:

```
+-----------------------------------+
| BYTES Buffer                      |
+-----------------------------------+
```

where:

The data packet is a *Buffer*. For the Read function, the *Buffer* must be large enough to hold requested data.

Remarks:

Function 64H performs the read operation on the device specified in this request. The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into *n* single-sector operations, and one sector at a time is read.

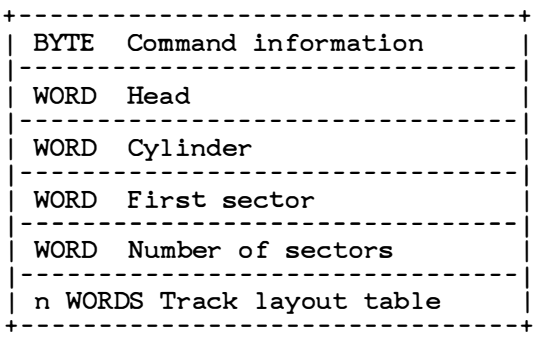
Note also that the device driver will *not* correctly read a non-512-byte sector if the read operation would generate a Dynamic Memory Allocation (DMA) violation error. Application writers must make sure that this error will *not* occur.

Function 65H: Verify Track

Purpose:

Function 65H verifies an operation on a drive.

Parameter Packet Format:



where:

The *Command information* field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1, and contains only consecutive sectors.
2–7	Reserved = 0

The *Head* field contains the physical head on the drive to perform the verify operation.

The *Cylinder* field contains the cylinder number for the verify.

The *First sector* field contains the logical sector number within the *Track layout table* to start the I/O operation.

Note

The *Track layout table* is based from zero (Origin 0), so the third sector is numbered 2.

The *Number of sectors* field contains the number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions will *not* step heads/tracks).

The *Track layout table* field is as follows:

```
+-----------------------------------+
| WORD  Sector number for sector 1  |
|-----------------------------------|
| WORD  Sector size for sector 1    |
|-----------------------------------|
| WORD  Sector number for sector 2  |
|-----------------------------------|
| WORD  Sector size for sector 2    |
|-----------------------------------|
| WORD  Sector number for sector 3  |
|-----------------------------------|
| WORD  Sector size for sector 3    |
+-----------------------------------+
                :
                :
+-----------------------------------+
| WORD  Sector number for sector n  |
|-----------------------------------|
| WORD  Sector size for sector n    |
+-----------------------------------+
```

The specified sector table provides information that is used during verify operations.

Remarks:

Function 65H performs the verify operation on the device specified in this request. The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into *n* single-sector operations, and one sector at a time is verified.

The Verify function does not use the Data Packet.

4.8 Physical Disk Control IOCtl Commands (Category 09H)

The physical disk Category 09H IOCtl commands are presented in the following order:

Function	Description
00H	Lock physical Drive
01H	Unlock physical Drive
44H	Physical Write Track
63H	Get Physical Device Parameters
64H	Physical Read Track
65H	Physical Verify Track

Category 09H is used to access physical, partitionable fixed disks. The unit field in the request packet that the device driver receives contains the physical disk number instead of unit number. The physical disk number relates back to a physical, partitionable disk that the device driver reported back to the system with Partitionable Fixed Disks (Command Code 23).

Note

For all Category 09H functions, the unit field in the Request Packet refers to the physical drive number that is supported by the device driver and *not* to the logical unit number.

The category 09H subfunctions relate to the entire partitionable fixed disk. Direct track and sector I/O begins at the beginning of the physical drive instead of at the beginning of an extended volume. Function 63H, Get Physical Device Parameters, describes the entire physical device.

Function 00H: Lock Physical Drive

Purpose:

Function 00H locks the physical drive and any of its associated logical units.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+---------------------------------------+
| DWORD   Set to zero                   |
+---------------------------------------+
```

Remarks:

Function 00H also affects the logical units on the physical drive.

Function 01H: Unlock Physical Drive

Purpose:

Function 01H unlocks the physical drive and any of its associated logical units.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD   Set to zero               |
+-----------------------------------+
```

Remarks:

Function 01H also affects the logical units on the physical drive.

Function 63H: Get Physical Device Parameters

Purpose:

Function 63H returns the device parameters for a physical device.

Parameter Packet Format:

BYTE	Command information

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

WORD	Reserved
WORD	Number of cylinders
WORD	Number of heads
WORD	Number of sectors/tracks
WORD	Reserved
WORD	Reserved
WORD	Reserved
WORD	Reserved

where:

Reserved is a reserved word set to zero.

Number of cylinders returns the number of cylinders on the physical drive.

Number of heads returns the number of heads on the physical drive.

Sectors per track returns the number of sectors per track on the physical drive.

Remarks:

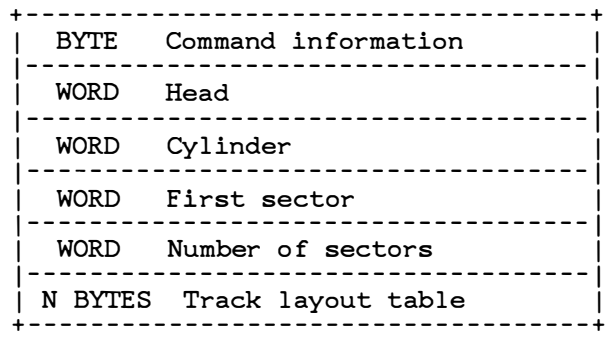
The values returned in the Data Packet apply to the entire physical disk.

Functions 44H: Physical Write Track

Purpose:

Function 44H writes to a physical track.

Parameter Packet Format:



where:

Command information field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1, and contains only consecutive sectors.
2–7	Reserved = 0

The *Head* field contains the physical head on the drive to perform the write operation.

The *Cylinder* field contains the cylinder number for the write.

The *First sector* field contains the logical sector number within the *Track layout table* to start the I/O operation.

Note

The sector layout table is based from zero (ORIGIN 0), so the third sector is numbered 2.

The *Number of sectors* field contains the number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions will *not* step heads/tracks).

The *Track layout table* field is as follows:

```
+-----------------------------------+
| WORD  Sector number for sector 1  |
|-----------------------------------|
| WORD  Sector size for sector 1    |
|-----------------------------------|
| WORD  Sector number for sector 2  |
|-----------------------------------|
| WORD  Sector size for sector 2    |
|-----------------------------------|
| WORD  Sector number for sector 3  |
|-----------------------------------|
| WORD  Sector size for sector 3    |
+-----------------------------------+
                :
                :
+-----------------------------------+
| WORD  Sector number for sector n  |
|-----------------------------------|
| WORD  Sector size for sector n    |
+-----------------------------------+
```

The specified sector table provides information that is used during write operations.

Data Packet Format:

```
+-----------------------------------+
| n BYTES Buffer                    |
+-----------------------------------+
```

where:

The Data Packet is an *n*-byte *Buffer*. For the Physical Write Track function, it contains the data to be written.

Remarks:

Function 44H performs the write operation on the device specified in the request. Its operation is similar to Category 08H, Function 44H, except that I/O is offset from the beginning of the physical drive instead of from the unit number (as in the Category 08H functions).

The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into *n* single-sector operations, and one sector at a time is written.

Functions 64H: Physical Read Track

Purpose:

Function 64H reads from a physical track.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE   Command information       |
|-----------------------------------|
|  WORD   Head                      |
|-----------------------------------|
|  WORD   Cylinder                  |
|-----------------------------------|
|  WORD   First sector              |
|-----------------------------------|
|  WORD   Number of sectors         |
|-----------------------------------|
| N BYTES  Track layout table       |
+-----------------------------------+
```

where:

Command information field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1, and contains only consecutive sectors.
2–7	Reserved = 0

The *Head* field contains the physical head on the drive to perform the read operation.

The *Cylinder* field contains the cylinder number for the read.

The *First sector* field contains the logical sector number within the *Track layout table* to start the I/O operation.

Note

The sector layout table is based from zero (ORIGIN 0), so the third sector is numbered 2.

The *Number of sectors* field contains the number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions will *not* step heads/tracks).

The *Track layout table* field is as follows:

```
+-----------------------------------+
| WORD  Sector number for sector 1  |
|-----------------------------------|
| WORD  Sector size for sector 1    |
|-----------------------------------|
| WORD  Sector number for sector 2  |
|-----------------------------------|
| WORD  Sector size for sector 2    |
|-----------------------------------|
| WORD  Sector number for sector 3  |
|-----------------------------------|
| WORD  Sector size for sector 3    |
+-----------------------------------+
                :
                :
+-----------------------------------+
| WORD  Sector number for sector n  |
|-----------------------------------|
| WORD  Sector size for sector n    |
+-----------------------------------+
```

The specified sector table provides information that is used during read operations.

Data Packet Format:

```
+-----------------------------------+
| n BYTES Buffer                    |
+-----------------------------------+
```

where:

The Data Packet is an *n*-byte *Buffer*. For the Physical Read Track function, the *Buffer* must be large enough to hold requested data.

Remarks:

Function 64H performs the read operation on the device specified in the request. Its operation is similar to Category 08H, Function 64H, except that I/O is offset from the beginning of the physical drive instead of from the unit number (as in the Category 08H functions).

The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into *N* single-sector operations, and one sector at a time is read.

Note also that the device driver will *not* correctly read a non-512-byte sector if the read operation would generate a DMA violation error. Application writers must make sure that this error will *not* occur. This function also affects the logical units on the physical drive.

Functions 65H: Physical Verify Track

Purpose:

Function 65H verifies I/O operations on a physical track.

Parameter Packet Format:

BYTE	Command information
WORD	Head
WORD	Cylinder
WORD	First sector
WORD	Number of sectors
N BYTES	Track layout table

where:

Command information field is a bitfield described as follows:

Bit	Description
0	Track layout contains nonconsecutive sectors or does not start with sector 1.
1	Track layout starts with sector 1, and contains only consecutive sectors.
2–7	Reserved = 0

The *Head* field contains the physical head on the drive to perform the verify operation.

The *Cylinder* field contains the cylinder number for the verify.

The *First sector* field contains the logical sector number within the *Track layout table* to start the I/O operation.

Note

The sector layout table is based from zero (ORIGIN 0), so the third sector is numbered 2.

The *Number of sectors* field contains the number of sectors to read (up to the maximum specified in the track table; the IOCtl subfunctions will *not* step heads/tracks).

The *Track layout table* field is as follows:

```
+-----------------------------------+
| WORD  Sector number for sector 1  |
|-----------------------------------|
| WORD  Sector size for sector 1    |
|-----------------------------------|
| WORD  Sector number for sector 2  |
|-----------------------------------|
| WORD  Sector size for sector 2    |
|-----------------------------------|
| WORD  Sector number for sector 3  |
|-----------------------------------|
| WORD  Sector size for sector 3    |
+-----------------------------------+
                :
                :
+-----------------------------------+
| WORD  Sector number for sector n  |
|-----------------------------------|
| WORD  Sector size for sector n    |
+-----------------------------------+
```

The specified sector table provides information that is used during verify operations.

Remarks:

Function 65H performs the verify operation on the device specified in the request. Its operation is similar to Category 08H, Function 65H, except that I/O is offset from the beginning of the physical drive instead of from the unit number (as in the Category 08H functions).

The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into *n* single-sector operations, and one sector at a time is verified.

4.9 Character Monitor IOCtl Commands (Category 0AH)

Function 40H: Register Monitor

Purpose:

Function 40H registers a monitor.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-------------------------------------+
| WORD   Position flag                |
|-------------------------------------|
| WORD   Index                        |
|-------------------------------------|
| DWORD  Address of input buffer      |
|-------------------------------------|
| WORD   Offset of output buffer      |
+-------------------------------------+
```

where:

Position flag is a Word that specifies the *PosFlag* parameter used in the **DosMonReg** API call. The *Position flag* may have one of the following values:

Value	Meaning
0	No positional preference
1	Front of the list
2	Back of the list

Index, a one-word field, contains a device-specific value.

Address of input buffer is a double-word field containing the address of the monitor input buffer that is initialized by the monitor dispatcher and used by the **DosMonRead** call.

Offset of output buffer is a Word that specifies the offset to the monitor output buffer that is initialized by the monitor dispatcher and used by the **DosMonWrite** call.

These fields are used by the device drivers to formulate the **MonRegister** calls (**DevHlp**).

Remarks:

None.

4.10 General Device Control IOCtl Commands (Category 0BH)

This section describes the three general device control subfunctions within Category 0BH of the Generic IOCtl commands. These three functions are:

Function	Description
01H	Flush Input Buffer
02H	Flush Output Buffer
60H	Query Monitor Support

Details of these functions are shown on the following pages.

Function 01H: Flush Input Buffer

Purpose:

Function 01H flushes the input buffer.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD Set to zero                 |
+-----------------------------------+
```

Remarks

None.

Function 02H: Flush Output Buffer

Purpose:

Function 02H flushes the output buffer.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD Set to zero                 |
+-----------------------------------+
```

Remarks:

None.

Function 60H: Query Monitor Support

Purpose:

Function 60H queries a device driver for monitor support.

Parameter Packet Format:

```
+-----------------------------------+
| BYTE Command information          |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| DWORD Set to zero                 |
+-----------------------------------+
```

Remarks:

If the device driver does not support character monitors, the device driver should return the system error "Monitors not supported" (error code 13H). If the device driver does support character monitors, it should return "No error" (error code 00H).

4.11 Old Command Interface

The old MS-DOS Request Packet interface is maintained only for a limited set of commands. The list of commands that are supported for character devices are as follows:

Code	Command
0	Init
3	IOCtl Read
4	Read (input)
5	Nondestructive Read No Wait
6	Input Status
7	Input Flush
8	Write (output)
9	Write with verify
10	Output Status
11	Output Flush
12	IOCtl Write
13	Device Open
14	Device Close
16	Generic IOCtl

Note that a real-mode application may access an old character device driver through the INT 21H interface as in MS-DOS 3.*x*. However, a protected-mode application may not access an old device driver.

MS-DOS 3.2 Generic IOCtl Support

There are two types of Generic IOCtl function calls supported in the MS OS/2 real mode for MS-DOS 3.2 applications.

- Function: AL=0DH

  This function works the same as it does in MS-DOS 3.2, with the addition that the register pair SI:DI is the address of the parameter block in MS OS/2, and DS:DX is the address of the data packet.

- Function: AL=0CH

  This is similar to function AL=0DH except that BX contains a handle to a device instead of to a drive letter. This is useful for character devices.

The register contents are as follows:

Register	Contents
AH	44H – IOCtl request
AL	0DH – Drive-oriented (for block devices)
BL	Drive number (for block devices)
AL	0CH – Handle-oriented (for character devices)
BX	Handle value (for character devices)
CH	Category
CL	Function
DS:DX	Data block
SI:DI	Parameter block

For more information about which categories and functions are supported, see the specific device IOCtl descriptions in this chapter.

Chapter 5
Device Helper Services

5.1 Device Helper Services

Many of the functions of an MS OS/2 device driver are related to system operations rather than to hardware operations. An interface to operating system services is therefore available to device drivers through the **DevHlp** interface.

Access to these system services is obtained at the time of device driver initialization. The request packet for the **Init** command contains a bimodal pointer to the **DevHlp** interface. The device driver does not have to be sensitive to the mode of operation before requesting **DevHlp** services.

A service is invoked by setting up the appropriate registers, loading a function code into the DL register, and making a FAR call to the **DevHlp** interface routine, whose address was supplied at device initialization time.

Table 5.1 lists the **DevHlp** services, their function codes, and a short description of each service.

Table 5.1
DevHlp Services and Function Codes

DevHlp Service	Code	Description
SchedClock	00H	Signal Clock Tick
DevDone	01H	Flag I/O Complete
Yield	02H	Yield CPU
TCYield	03H	Yield CPU to Time-critical Thread
Block	04H	Block Thread from Running
Run	05H	Release Blocked Thread
SemRequest	06H	Request Semaphore
SemClear	07H	Clear Semaphore
SemHandle	08H	Get Semaphore Handle
PushReqPacket	09H	Push Request Packet onto Queue
PullReqPacket	0AH	Pull Request Packet from Queue
PullParticular	0BH	Pull Specific Request Packet from Queue
SortReqPacket	0CH	Insert Request Packet in Sorted Order
AllocReqPacket	0DH	Allocate Request Packet
FreeReqPacket	0EH	Free Allocated Request Packet

QueueInit	0FH	Initialize Character Queue
QueueFlush	10H	Flush Character Queue
QueueWrite	11H	Insert Character in Queue
QueueRead	12H	Read Character from Queue
Lock	13H	Lock Memory Segment
Unlock	14H	Unlock Memory Segment
PhysToVirt	15H	Map Physical Address to Virtual Address
VirtToPhys	16H	Map Virtual Address to Physical Address
PhysToUVirt	17H	Map Physical Address to User Virtual Address
AllocPhys	18H	Allocate Physical Memory
FreePhys	19H	Free Physical Memory
SetROMVector	1AH	Set ROM BIOS Interrupt Handler
SetIRQ	1BH	Set Hardware Interrupt Handler
UnSetIRQ	1CH	Reset Hardware Interrupt Handler
SetTimer	1DH	Set Timer Handler
ResetTimer	1EH	Reset Timer Handler
MonitorCreate	1FH	Create Monitor
Register	20H	Register Monitor
DeRegister	21H	Deregister Monitor
MonWrite	22H	Write Data Records to Monitor
MonFlush	23H	Flush Data from Monitor Stream
GetDOSVar	24H	Get Pointer to DOS Variable
SendEvent	25H	Send Event
ROMCritSection	26H	Flag Critical Section of Execution
PortUsage	27H	Indicate I/O Port Usage
GrantPortAccess	28H	Grant Access to I/O Ports
VerifyAccess	29H	Verify Memory Access
EOI	31H	Issue End-Of-Interrupt
UnPhysToVirt	32H	Mark Completion of Virtual Address Use
TickCount	33H	Modify Timer

As discussed in Chapter 2, "Device Driver Architecture," device driver code may run in one of four contexts:

- Kernel mode — the context in which the device driver strategy routine runs.
- Interrupt mode — the context in which the device driver hardware-interrupt handler runs.
- User mode — the context in which the device driver handler for a real-mode ROM BIOS interrupt runs.
- Initialization mode — the context in which the device driver strategy routine runs when it is called by the **Init** request packet.

Certain restrictions apply to when individual **DevHlp** services may be used. Table 5.2 outlines which **DevHlp** services are allowed in which contexts (kernel, interrupt, user, or initialization).

Table 5.2

DevHlp Services and Corresponding States

DevHlp Service	Code	Kernel	Interrupt	User	Initialization
SchedClock	00H		*		
DevDone	01H	*	*		
Yield	02H	*			
TCYield	03H	*			
Block	04H	*		*	
Run	05H	*	*	*	
SemRequest	06H	*		*	
SemClear	07H	*	*	*	
SemHandle	08H	*	*		
PushReqPacket	09H	*			
PullReqPacket	0AH	*	*		
PullParticular	0BH	*	*		
SortReqPacket	0CH	*			
AllocReqPacket	0DH	*			
FreeReqPacket	0EH	*			
QueueInit	0FH	*	*	*	
QueueFlush	10H	*	*	*	
QueueWrite	11H	*	*	*	
QueueRead	12H	*	*	*	
Lock	13H	*			*
Unlock	14H	*			*
PhysToVirt	15H	*	*		*
VirtToPhys	16H	*			*
PhysToUVirt	17H	*			
AllocPhys	18H	*			*
FreePhys	19H	*			*
SetROMVector	1AH	*			*
SetIRQ	1BH	*			*
UnSetIRQ	1CH	*	*		*
SetTimer	1DH	*			*
ResetTimer	1EH	*			*

Table 5.2 *(continued)*

DevHlp Service	Code	Kernel	Interrupt	User	Initialization
MonitorCreate	1FH	*			*
Register	20H	*			
DeRegister	21H	*			
MonWrite	22H	*	*	*	
MonFlush	23H	*			
GetDOSVar	24H	*			*
SendEvent	25H	*	*		
ROMCritSection	26H			*	
PortUsage	27H	*			*
GrantPortAccess	28H	*			
VerifyAccess	29H	*			
EOI	31H		*		*
UnPhysToVirt	32H	*	*		*
TickCount	33H	*	*	*	*

5.2 Categorical Listing of Device Helper Services

Calling conventions for each of the helper routines follow. In addition to the returns noted under each routine, the interrupt flag may be set or cleared by some routines, and other flags may be affected by the calls. Some routines require that the interrupt flag be off when they are called.

The **DevHlp** services are listed in the following categories (section numbers are in parentheses).

System Clock Management (Section 5.3)

SchedClock

Process Management (Section 5.4)

Block
DevDone
Run
TCYield
Yield

Semaphore Management (Section 5.5)

SemClear
SemHandle
SemRequest

Request Queue Management (Section 5.6)

AllocReqPacket
FreeReqPacket

PullParticular
PullReqPacket
PushReqPacket
SortReqPacket

Character Queue Management (Section 5.7)

QueueFlush
QueueInit
QueueRead
QueueWrite

Memory Management (Section 5.8)

AllocPhys
FreePhys
Lock
PhysToUVirt
PhysToVirt
Unlock
UnPhysToVirt
VerifyAccess
VirtToPhys

Interrupt Management (Section 5.9)

EOI
SetIRQ
SetROMVector
UnSetIRQ

Timer Services (Section 5.10)

ResetTimer
SetTimer
TickCount

Monitor Management (Section 5.11)

DeRegister
MonFlush
MonitorCreate
MonWrite
Register

System Services (Section 5.12)

GetDOSVar
GrantPortAccess
PortUsage
ROMCritSection
SendEvent

5.3 System Clock Management

SchedClock: Signal Clock Tick

Purpose:

The **SchedClock** function is an entry point called on each occurrence of a clock tick.

Calling Sequence:

```
MOV  AL,Millisecs          ;Milliseconds since last call
MOV  DH,EOIFlag            ;Indicator of EOI
MOV  DL,DEVHLP_SCHEDCLOCK
CALL [Device_Help]
```

where:

Millisecs is the number of milliseconds elapsed since the last call.

EOIFlag is a flag indicating end-of-interrupt (EOI) as follows:

Value	Meaning
0	Prior to EOI
1	After EOI

Returns:

None.

Remarks:

SchedClock is called by the clock device driver to indicate the passage of system time. The clock tick is then dispersed to the appropriate components of the system.

The clock device driver's interrupt handler must run with interrupts enabled, prior to issuing the EOI for the timer interrupt. Any critical processing, such as updating the fraction-of-seconds count, must be done prior to calling the **SchedClock** function. **SchedClock** must then be called to allow system processing prior to the dismissal of the interrupt. When **SchedClock** returns, the clock device driver must issue the EOI and call **SchedClock** again. Note that once the EOI has been issued, the device driver's interrupt handler may be reentered. **SchedClock** is also reentrant.

5.4 Process Management

This section describes the device helper functions used for process management. These functions are listed as follows:

Function	Description
Block	Block Thread from Running
DevDone	Flag I/O Complete
Run	Release Blocked Thread
TCYield	Yield CPU to Time-critical Thread
Yield	Yield CPU

Block: Block Thread from Running

Purpose:

The **Block** function "sleeps" the current device driver thread until either the **Run** function is issued on the event identifier or a time-out occurs.

Calling Sequence:

```
MOV  BX,EventIdLow          ;Low word of event id
MOV  AX,EventIdHigh         ;High word of event id
MOV  DI,TimeLimitHigh       ;Time-out interval
MOV  CX,TimeLimitLow        ;in milliseconds
MOV  DH,InterruptibleFlag   ;Tells if sleep is interruptible
MOV  DL,DEVHLP_BLOCK
CALL [Device_Help]
```

where:

EventIdLow is the low word of the event identifier.

EventIdHigh is the high word of the event identifier.

TimeLimitHigh and *TimeLimitLow* indicate the limits of the time-out interval in milliseconds. If the value is –1, a time-out will never occur.

InterruptibleFlag is a flag that specifies whether or not the "sleep" is interruptible. *InterruptibleFlag* has one of the following values:

Value	Meaning
0	Sleep is interruptible
1	Sleep is noninterruptible

Returns:

C —clear if event wakeup, set if unusual wakeup.

Z —set if wakeup due to time-out, cleared if sleep was interrupted.

AL —Awake code, nonzero if unusual wakeup.

- Interrupts enabled.

Remarks:

As a side-effect, this function switches context.

The return from the **Block** function indicates, through the condition codes, whether the "wakeup" occurred from a normal **Run** call or from an expiration of the time limit.

MS OS/2 will not immediately return from **Block**. Instead, it removes the current thread from the run queue and starts executing some other thread. When the time limit expires, the original thread is reactivated and **Block** returns.

Like **Block**, **Run** is called with the same *event identifier* (an arbitrary 32-bit value) or when the time limit expires. One MS OS/2 convention must be followed, however, to coordinate with the thread issuing the **Run** function. The standard convention for **Block/Run** operations is to use the address of some structure or memory cell associated with the reason for blocking and running. For example, a thread that blocks until some resource is cleared normally blocks "on" the address of the ownership flag for that resource.

Since bimodal device drivers may be blocked in one mode and run in the other, using the virtual address as the event identifier is not sufficient. Also, the logical address of an item in one mode is not the same as it would be in the other mode. A possible option for device drivers is to use the following convention: drivers can **Block/Run** on the 32-bit physical address of the object.

The **Block/Run** mechanism cannot guarantee immunity from a random wakeup by another thread running a 32-bit key value that just happens to match some unrelated blocking thread's key. The goal is to choose keys that are known to the "Blocker" and the "Runner" and that have a high likelihood of uniqueness. Users of **Block/Run** must always check the reason for their wakeup to make sure that the event really took place and that the wakeup wasn't accidental.

It is important to use the following sequence when calling **Block**:

```
Disable Interrupts
while (need to wait)
      Block(value)
      Disable Interrupts
```

To avoid a deadlock by getting an interrupt-time **Run** call before completing the call to **Block**, interrupts are turned off *before* checking the condition (for example, "I/O done" or "Resource freed"). **Block** then reenables the interrupts. Also note the "while" clause, which is essential to recheck the awaited condition and, if necessary, to re-disable interrupts and re-call **Block**. The convention of using an address as an event identifier should prevent double use of an identifier.

A time limit of –1 means that **Block** waits indefinitely until **Run** is called. **Block** can only be called by the task-time portion of a device driver.

When using **Block** to block a thread, the driver can specify whether the sleep may be interrupted. If the sleep is interruptible, MS OS/2 can abort the blocked thread and return from the **Block** without using a corresponding **Run**. In general, the sleep should be marked as interruptible, unless the sleep duration is expected to be less than a second.

Block exits to the caller when the wakeup is issued, when the time limit expires, or when the thread is signaled,

The return from the **Block** must be checked to determine the nature of the wakeup. If the return from the **Block** indicates that the sleep was interrupted, then some internal event occurred that requires attention (such as a signal, the death of a process, or some other forced action). The device driver should respond by performing any necessary cleanup, setting the error code in the *Status* field of the request packet, then returning the request packet to the kernel via the **DevDone** function.

See Also:

Run

DevDone: Flag I/O Complete

Purpose:

The **DevDone** function signals that a request has completed.

Calling Sequence:

```
LES  BX,RequestPacket          ;Pointer to I/O request packet.
MOV  DL,DEVHLP_DEVDONE
CALL [Device_Help]
```

where:

RequestPacket contains a pointer to an I/O request packet.

Returns:

None.

Remarks:

Since the virtual address of a request packet is valid in both real and protected mode, the device driver may pass the request packet pointer to **DevDone** without being sensitive to the mode.

DevDone is typically called from the device interrupt routine. The device driver should set any error flags in the *Status* field of the request packet before calling the routine.

DevDone does not apply to request packets allocated by the **AllocReqPacket** function call.

See Also:

AllocReqPacket

Run: Release Blocked Thread

Purpose:

Run, the companion function to **Block**, releases a blocked thread.

Calling Sequence:

```
MOV  BX,EventIdLow          ;Low word of event id
MOV  AX,EventIdHigh         ;High word of event id
MOV  DL,DEVHLP_RUN
CALL [Device_Help]
```

where:

EventIdLow is the low word of the event identifier for which threads are being wakened.

EventIdHigh is the high word of the event identifier.

Returns:

None.

Remarks:

When **Run** is called (often at interrupt time), it awakens *all* threads that were blocked for this particular event identifier, then returns immediately to its caller. The awakened threads will be run at the next available opportunity. For a more detailed discussion of the process, see the **Block** function.

See Also:

Block

TCYield: Yield CPU to Time-critical Thread

Purpose:

The **TCYield** function is similar to the **Yield** function, except that the CPU may be yielded only to a time-critical thread, if one is available.

Calling Sequence:

```
MOV DL,DEVHLP_TCYIELD
CALL [Device_Help]
```

Returns:

None.

Remarks:

The device driver need not perform both a **Yield** and a **TCYield**, since the **TCYield** function is a subset of the **Yield** function.

For best performance, the device driver should check the *TCYieldFlag* once every three milliseconds. If the flag is set, the device driver should call the **TCYield** function to yield the CPU to a time-critical thread.

The address of the *TCYieldFlag* is obtained from the **GetDOSVar** call.

See Also:

GetDosVar, Yield

Yield: Yield CPU

Purpose:

The **Yield** function yields the CPU to a thread of equal or higher priority, if one is scheduled.

Calling Sequence:

```
MOV DL,DEVHLP_YIELD
CALL [Device_Help]
```

Returns:

None.

Remarks:

Device drivers are the part of the kernel that can take a lot of CPU time, especially those drivers that perform program I/O on long strings of data or that poll the device. These drivers should periodically check the *YieldFlag* and call the **Yield** function to yield the CPU if another process needs it. Much of the time the context won't switch; **Yield** switches context only if an equal or higher priority thread is scheduled to run.

The address of the *YieldFlag* is obtained from the **GetDOSVar** call.

Note

If you want to yield the CPU for more than 2.5 milliseconds, you should use the **TCYield** function. The **Yield** function should be used for high-priority device drivers if they will hold the CPU for more than 50 milliseconds, or for all device drivers that poll their devices.

The device driver need not perform both a **Yield** and a **TCYield**, since the **Yield** function is a superset of the **TCYield** function.

The DOS is designed so that the CPU is never preemptively scheduled while in kernel mode. In general, the kernel code either performs its job and exits quickly, or it blocks waiting for I/O (or a resource).

For best performance, the device driver should check the *YieldFlag* once every three milliseconds. If the flag is set, the device driver should call **Yield**.

See Also:

TCYield

5.5 Semaphore Management

This section describes the device helper functions for managing semaphores. These functions are listed as follows:

Function	Description
SemClear	Release Semaphore
SemHandle	Get Semaphore Handle
SemRequest	Claim Semaphore

There are two kinds of semaphores, *RAM* semaphores and *system* semaphores. RAM semaphores are defined by the semaphore user by allocating a double-word of storage and by using the address in the semaphore calls. MS OS/2 provides no resource management on RAM semaphores (such as freeing the semaphore when the owner terminates). System semaphores are created by an application-level process through a system call. MS OS/2 provides full resource management on system semaphores, including freeing and notification of when the owner terminates.

Typically, a device driver will create and use its own RAM semaphores to control operations within itself and use a client's system semaphores to communicate with the client. Note that there is no service that allows a device driver to create its own system semaphores.

For RAM semaphores, the device driver:

- Must initialize the RAM semaphore double-word to zero prior to use
- May use RAM semaphores in user mode

For system semaphores, the device driver:

- Must obtain a semaphore handle with the **SemHandle** function
- Must use the **SemHandle** service to indicate that the system semaphore is "in-use" or "not-in-use"

"In-use" means that the device driver may be referencing the system semaphore. "Not-in-use" means that the device driver has finished using the system semaphore and will not reference it again. For details, see the **SemHandle** function in this section.

SemClear: Release Semaphore

Purpose:

The **SemClear** function, used either at task time or interrupt time, releases a semaphore and restarts any blocked threads waiting on the semaphore.

Calling Sequence:

```
MOV  BX,SemHandleLow          ;Semaphore handle
MOV  AX,SemHandleHigh         ;
MOV  DL,DEVHLP_SEMCLEAR
CALL [Device_Help]
```

where:

SemHandleLow and *SemHandleHigh* specify the low and high words of the semaphore handle for the semaphore to be cleared.

Returns:

C —clear if no error, set if error.

AX = Error code

Remarks:

A device driver may clear either a RAM semaphore or a system semaphore.

The handle for a RAM semaphore is the *virtual address* of the double-word of storage allocated for the semaphore. Virtual address is a generic term used for addresses: segment:offset for real mode, selector:offset for protected mode.

If the device driver references the RAM semaphore at interrupt time, it must manage the addressability to the RAM semaphore.

For a system semaphore, the handle must be passed to the device driver by the caller via a Generic IOCtl call. By using the **SemHandle** function, the device driver must convert the caller's handle to a system handle.

See Also:

SemHandle

SemHandle: Get Semaphore Handle

Purpose:

The **SemHandle** function provides a semaphore handle to the device driver.

Calling Sequence:

```
MOV  BX,SemHandleLow      ;Semaphore identifier
MOV  AX,SemHandleHigh     ;
MOV  DH,UsageFlag         ;Indicates if in use
MOV  DL,DEVHLP_SEMHANDLE
CALL [Device_Help]
```

where:

SemHandleLow and *SemHandleHigh* specify the low and high words of the semaphore handle.

UsageFlag is a flag that indicates whether or not the requested semaphore is in use. *UsageFlag* has one of the following values:

Value	Meaning
0	Not-in-use
1	In-use

Returns:

C —clear if no error.

AX:BX is set to the system handle for the semaphore.

C —set if error.

AX = Error code:

- Invalid semaphore handle.

Remarks:

This function converts the semaphore handle (or user *key*), provided by the caller of the device driver, to a system handle that the device driver may use. This handle then becomes the key that the device driver uses to reference the system semaphore, allowing the system semaphore to be referenced at interrupt time by the device driver. The device driver also uses this key when it is finished with the system semaphore: To indicate that it is finished with the system semaphore, the device driver must call **SemHandle** with the *UsageFlag* which indicates when the device driver is finished.

SemHandle is called at task time to indicate that the system semaphore is "in-use," and is called at either task time or interrupt time to indicate that the system semaphore is "not-in-use." "In-use" means that the device driver may be referencing the system semaphore. "Not-in-use" means that the device driver has finished using the system semaphore and will not be referencing it again.

The key of a RAM semaphore is its *virtual address*, where virtual address is the generic term for both real- and protected-mode address forms (segment:offset, selector:offset). **SemHandle** may be used for RAM semaphores, but since RAM semaphores have no system handles, **SemHandle** will simply return the RAM semaphore key back to the caller.

It is necessary to call **SemHandle** at task time to set a system semaphore "in-use" because:

- The caller-supplied semaphore handle refers to task-specific system semaphore structures. These structures are not available at interrupt time, so **SemHandle** converts the task-specific handle to a system-specific handle. For uniformity, the other semaphore **DevHlp** functions accept only system-specific handles, regardless of the mode.

- An application could delete a system semaphore while the device driver is using it. If a second application were to create a system semaphore soon after, the system structure that had been used by the original semaphore could be reassigned. A device driver that then tried to manipulate the original process's semaphore would inadvertently manipulate the new process's semaphore. For this reason, the **SemHandle** "in-use" indicator increments a counter so that even though the calling thread may still delete its task-specific reference to the semaphore, the semaphore still remains in the system structures.

A device driver must subsequently call **SemHandle** with "not-in-use" when it has finished using the semaphore, so that the system semaphore structure can be freed. For every call indicating "not-in-use," there must be a matching call indicating "in-use."

SemRequest: Claim Semaphore

Purpose:

The **SemRequest** function claims a semaphore.

Calling Sequence:

```
MOV  BX,SemHandleLow      ;Semaphore handle
MOV  AX,SemHandleHigh     ;
MOV  CX,SemTimeoutLow     ;Time-out value
MOV  DI,SemTimeoutHigh    ;in milliseconds
MOV  DL,DEVHLP_SEMREQUEST
CALL [Device_Help]
```

where:

SemHandleLow and *SemHandleHigh* are the low and high words of the semaphore handle being claimed.

SemTimeoutLow and *SemTimeoutHigh* specify the values for the time-out limit which may be one of the following:

Value	Meaning
–1	Wait forever, time-out never occurs
0	No Wait if semaphore is owned
0	Time-out limit

Returns:

C —cleared if no error, set if error.

AX = Error code:

- Semaphore time-out.
- Interrupt.
- Semaphore owner died.

- Invalid handle.
- Too many semaphore requests.

Remarks:

SemRequest checks the status of a semaphore. If it is unowned, then **SemRequest** sets it as owned and returns immediately to the caller. If the semaphore is owned, **SemRequest** will optionally block the device driver thread until the semaphore is released or until a time-out occurs, then try again. The time-out parameter places an upper bound on the amount of time for **SemRequest** to block before returning to the requesting device driver thread.

The semaphore handle for a RAM semaphore is the *virtual address* of the double-word of storage allocated for the semaphore. Virtual address is a generic term used for addresses: segment:offset for real mode, selector:offset for protected mode.

If the device driver references the RAM semaphore at interrupt time, it must manage the addressability to the RAM semaphore. But for a system semaphore, the handle must be passed to the device driver by the caller via a Generic IOCtl call. By using the **SemHandle** function, the device driver must then convert the caller's handle to a system handle.

Note that the **SemRequest** function is valid in user mode only for RAM semaphores. System semaphores are not available for use in user mode by device drivers.

See Also:

SemHandle

5.6 Request Queue Management

The functions described in this section are listed as follows, along with a brief description of each:

Function	Description
AllocReqPacket	Allocate Request Packet
FreeReqPacket	Free Allocated Request Packet
PullParticular	Pull Specific Request Packet from Queue
PullReqPacket	Pull Request Packet from Queue
PushReqPacket	Push Request Packet onto Queue
SortReqPacket	Insert Request in Sorted Order

These functions provide simple linked-list management that allows device drivers to easily maintain a list or queue of request packets to be serviced. Prior to using these functions, the device driver must allocate and initialize a Dword queue header to zero.

Typical use of these functions is for the device driver to queue request packets that cannot be serviced immediately due to the device being busy. For example, a disk device driver that supports more than one device would maintain a request packet chain for each device.

AllocReqPacket: Allocate Request Packet

Purpose:

The **AllocReqPacket** function returns a pointer to a request packet.

Calling Sequence:

```
MOV  DH, WaitFlag              ;Wait for available request packet
MOV  DL,DEVHLP_ALLOCREQPACKET
CALL [Device_Help]
```

where:

WaitFlag is a flag that specifies whether to wait for an available request packet. *WaitFlag* has one of the following values:

Value	Meaning
0	Wait for request packet
1	No Wait, return immediately

Returns:

C —cleared if a request packet was allocated,

ES:BX is set to the address of the allocated request packet.

C —set if a request packet was not allocated.

Remarks:

AllocReqPacket returns a bimodal pointer to a maximal-sized request packet. The bimodal pointer is a virtual address that is valid for both real and protected modes.

Some device drivers, notably the disk device driver, need to have additional request packets to service task-time requests. Since device drivers are bimodal, they cannot use a packet residing in their data segment because the resulting pointer is not bimodal.

Request packets that were allocated by an **AllocReqPacket** can be placed in the request packet queue. Request packets allocated in this manner should be returned to the kernel as soon as possible via the **FreeReqPacket** function. As a whole, the system has a limited number of request packets, so it is important that a device driver not allocate request packets and hold them for future use.

See Also:

FreeReqPacket

FreeReqPacket: Free Allocated Request Packet

Purpose:

The **FreeReqPacket** function releases a request packet previously allocated by the **AllocReqPacket** function.

Calling Sequence:

```
LES  BX,RequestPacket          ;Pointer to previous request packet
MOV  DL,DEVHLP_FREEREQPACKET
CALL [Device_Help]
```

where:

RequestPacket is a pointer to the request packet that was requested previously.

Returns:

None.

Remarks:

The device driver should free a request packet only if it has been allocated previously by the **AllocReqPacket** function. The **DevDone** function should not be used to return an allocated request packet.

The system has a limited number of request packets, so it is important that a device driver not allocate request packets and hold them for future use.

See Also:

AllocReqPacket, DevDone

PullParticular: Pull Specific Request Packet from Queue

Purpose:

The **PullParticular** function pulls the specified packet address from the selected device queue.

Calling Sequence:

```
LDS  SI,[Queue]                ;Head of device list
LES  BX,RequestPacket          ;Pointer to device request packet
MOV  DL,DEVHLP_PULLPARTICULAR
CALL [Device_Help]
```

where:

Queue is a pointer to the Dword that specifies the head of the device list. It specifies the next request to perform and should match the **PushReqPacket** value.

RequestPacket is a pointer to the device request packet.

Returns:

C —clear if no error, set if the specified request packet is not found.

Remarks:

PullParticular can be used to remove request packets that were allocated by an **AllocReqPacket** from the request packet queue.

See Also:

AllocReqPacket, PushReqPacket

PullReqPacket: Pull Request Packet from Queue

Purpose:

The **PullReqPacket** function pulls the next waiting request packet address from the selected device queue.

Calling Sequence:

```
LDS  SI,[Queue]                ;Head of device list
MOV  DL,DEVHLP_PULLREQPACKET
CALL [Device_Help]
```

where:

Queue is a pointer to the Dword that specifies the head of the device list. It specifies the next request to perform and should match the **PushReqPacket** value.

Returns:

C —clear if no error.

ES:BX Pointer to device packet.

C —set if there is no request packet.

Remarks:

Typically, a device driver uses the **PushReqPacket** and **PullReqPacket** functions to maintain a request queue for each of its devices. Queue elements are chained onto the pointer to the request packet.

PullReqPacket can also remove request packets that were allocated by an **AllocReqPacket** from the request packet queue.

If there is no request packet (such as when characters are typed on the keyboard before they are read), an indicator is set on return.

See Also:

AllocReqPacket, PullReqPacket

PushReqPacket: Push Request Packet onto Queue

Purpose:

The **PushReqPacket** function adds the current device request packet to the list of packets that will be executed by the device driver.

Calling Sequence:

```
LDS  SI,[Queue]                ;Pointer to head of device list
LES  BX,RequestPacket          ;Pointer to device request packet
MOV  DL,DEVHLP_PUSHREQPACKET
CALL [Device_Help]
```

where:

Queue is a pointer to the Dword that specifies the head of the device list. It specifies the next request to perform.

RequestPacket is a pointer to the device request packet.

Returns:

None.

Remarks:

The device driver task-time thread should add all incoming read/write requests to its request list. This thread should then determine whether the interrupt-time thread is active, and if not, should send the request to the device. Since the device may be active at this point, the task-time thread must turn off interrupts before calling the device (otherwise, a window exists in which the device finishes before the packet is put on the list).

PushReqPacket may be used to place request packets allocated by an **AllocReqPacket** in the request packet queue.

See Also:

AllocReqPacket, PullReqPacket

SortReqPacket: Insert Request Packet in Sorted Order

Purpose:

The **SortReqPacket** function, used by block (disk) device drivers, adds a new request to the block drivers' work queue, inserting the request packet in the order of its starting sector number.

Calling Sequence:

```
LDS  SI,[Queue]                ;Pointer to head of device list
LES  BX,RequestPacket          ;Pointer to device request packet.
MOV  DL,DEVHLP_SORTREQPACKET
CALL [Device_Help]
```

where:

Queue is a pointer to the Dword that specifies the head of the device list. It specifies the next request to perform and should be initialized to zero.

RequestPacket is a pointer to the device request packet.

Returns:

None.

Remarks:

Sorting by sector number reduces the length and number of head seeks. This is a simple algorithm and does not account for multiple heads on the media or for the target drive in the request packet. **SortReqPacket** inserts the current request packet into the specified queue of packets in sorted order according to its (the packet's) starting sector number.

The **SortReqPacket** function can be used to place request packets that were allocated by a call to the **AllocReqPacket** function in the request packet queue.

5.7 Character Queue Management

The following list briefly describes the device helper functions used for managing the character queue. These functions are described in more detail in the remainder of this section.

Function	Description
QueueFlush	Clear Character Queue
QueueInit	Initialize Character Queue
QueueRead	Read Character from Queue
QueueWrite	Insert Character into Queue

The character queue management functions operate on a simple circular buffer of characters, based on the following *CharQueue* queue header:

Length	Parameter
WORD	*Qsize*
WORD	*Qchrout*
WORD	*Qcount*
BYTE	*Qbase*

where:

Qsize is the size of queue in bytes.

Qchrout is the index of the next character out.

Qcount is the count of the characters in the queue.

Qbase indicates the start of queue buffer.

Prior to using the character queue functions, a device driver must allocate the queue header, initialize the *Qsize* field, and call **QueueInit** before calling the other functions. The other fields in the queue header are managed by the queue services and need not be examined or altered by callers of the services.

Typically, the character device drivers use the character queues to buffer data. Since the device driver always has addressability to its data segment, there is no need to maintain addressability of the queue header as a 32-bit physical address.

QueueFlush: Clear Character Queue

Purpose:

The **QueueFlush** function clears the specified character queue structure.

Calling Sequence:

```
MOV  BX,OFFSET DS:[Queue]; Points to the queue structure
MOV  DL,DEVHLP_QUEUEFLUSH
CALL [Device_Help]
```

where:

Queue is a pointer to the queue structure to be flushed. The *Qsize* field of the character queue must be set up.

Returns:

None.

QueueInit: Initialize Character Queue

Purpose:

The **QueueInit** function initializes the specified character queue structure.

Calling Sequence:

```
MOV  BX,OFFSET DS:[Queue]; Points to the queue structure
MOV  DL,DEVHLP_QUEUEINIT
CALL [Device_Help]
```

where:

Queue points to the queue structure to be initialized. The *Qsize* field must be set up.

Returns:

None.

Remarks:

QueueInit must be called before any other queue manipulation subroutine. The *Qsize* field must be initialized before calling **QueueInit**.

QueueRead: Read Character from Queue

Purpose:

The **QueueRead** function returns and removes a character from the beginning of the specified character queue buffer.

Calling Sequence:

```
MOV  BX,OFFSET DS:[Queue]; Points to the queue structure
MOV  DL,DEVHLP_QUEUEREAD
CALL [Device_Help]
```

where:

Queue is a pointer to the queue structure where a character will be returned and removed.

Returns:

C —set if the queue is empty, cleared otherwise.

AL = the character read from the queue.

QueueWrite: Insert Character into Queue

Purpose:

The **QueueWrite** function inserts a character at the end of the specified character queue buffer.

Calling Sequence:

```
MOV  BX,OFFSET DS:[Queue]; Points to the queue structure
MOV  AL,Character          ;Character to insert on queue
MOV  DL,DEVHLP_QUEUEWRITE
CALL [Device_Help]
```

where:

Queue is a pointer to the queue structure where a character will be inserted.

Character is the character to be inserted at the end of the queue.

Returns:

C —set if queue is full, clear if character stored successfully.

5.8 Memory Management

This section describes the device helper functions for memory management. These functions are listed as follows:

Function	Description
AllocPhys	Allocate Physical Memory
FreePhys	Free Physical Memory
Lock	Lock Memory Segment
PhysToUVirt	Map Physical Address to User Virtual Address
PhysToVirt	Map Physical Address to Virtual Address
Unlock	Unlock Memory Segment
UnPhysToVirt	Mark Completion of Virtual Address Use
VerifyAccess	Verify Memory Access
VirtToPhys	Map Virtual Address to Physical Address

AllocPhys: Allocate Physical Memory

Purpose:

The **AllocPhys** function allocates a block of fixed memory.

Calling Sequence:

```
MOV  BX,SizeLow              ;32-bit block size in bytes
MOV  AX,SizeHigh             ;
MOV  DH,HighOrLow            ;Relative position to 1 megabyte
MOV  DL,DEVHLP_ALLOCPHYS
CALL [Device_Help]
```

where:

SizeLow and *SizeHigh* are the low and high words of the 32-bit block size in bytes.

HighOrLow specifies the position of the allocated memory relative to the one-megabyte memory boundary. It may be one of the following:

Value	Meaning
0	Allocated memory is above one megabyte
1	Allocated memory is below one megabyte

Returns:

C —clear if memory allocated.

AX:BX = 32-bit physical address

C —set if error.

AX = Error code:

- Memory not allocated.

Remarks:

The memory allocated by this function is fixed memory, and may not be moved by the **Unlock** call.

If you attempt to allocate memory above one megabyte, and if no memory above one megabyte is available, an error is returned. The device driver could then attempt to allocate low memory.

Conversely, if you attempt to allocate memory below one megabyte, and if no memory below one megabyte is available, an error is returned. The device driver could then attempt to allocate high memory.

See Also:

Unlock

FreePhys: Free Physical Memory

Purpose:

The **FreePhys** function releases memory allocated by the **AllocPhys** call.

Calling Sequence:

```
MOV  BX,AddressLow          ;32-bit physical address
MOV  AX,AddressHigh         ;
MOV  DL,DEVHLP_FREEPHYS
CALL [Device_Help]
```

where:

AddressLow and *AddressHigh* are the low and high words of the 32-bit physical address of a memory segment that was specified in a previous **AllocPhys** call.

Returns:

C —clear if memory freed, set if error.

AX = Error code:

- Cannot free memory not previously allocated with **AllocPhys**.

Remarks:

Any memory that the device driver allocated with the **AllocPhys** call should be released prior to device driver termination.

See Also:

AllocPhys

Lock: Lock Memory Segment

Purpose:

The **Lock** function, which is called by device drivers at task time, locks a memory segment.

Calling Sequence:

```
MOV  AX,Segment          ;Selector or segment
MOV  BH,Duration         ;Duration of lock
MOV  BL, WaitFlag        ; Wait or return
MOV  DL,DEVHLP_LOCK
CALL [Device_Help]
```

where:

Segment is the selector or the segment to be locked.

Duration specifies the duration of the lock with one of the following values:

Value	Meaning
0	Short-term lock
1	Long-term lock

WaitFlag is a flag specifying one of the following:

Value	Meaning
0	Wait (block) until memory segment is locked
1	No wait, return if not immediately available

Returns:

C —clear if segment locked.

AX:BX = Lock handle.

C —set if segment unavailable or invalid handle.

Remarks:

If the called segment is unavailable, the caller must specify whether **Lock** should either return immediately, or block until the segment is available and locked.

If the lock duration parameter indicates that the segment is expected to be locked (fixed) for several seconds or longer, the segment may be moved to the region reserved for fixed, protected-mode segments.

Warning

The duration of the lock should be set to zero (short-term) for operations that are expected to complete in two seconds or less. Using short-term locks for longer periods of time can prevent an adequate amount of movable, swappable memory from being available for system use, thus hanging up the system.

Note that the **Lock** call need only be done on addresses that are received from user processes (such as in the case of an address that is passed via an IOCtl call).

Note also that prior to requesting the **Lock**, the device driver must verify the user process's access to the memory by using the **VerifyAccess** device helper call. The device driver must not yield the CPU between the **VerifyAccess** and the **Lock**; otherwise, the user process could shrink the segment before it has been locked. Once the user access has been verified, the device driver may convert the virtual address to a physical address and lock the memory. The access verification is then valid for the duration of the lock.

See Also:

Unlock, VerifyAccess

PhysToUVirt: Map Physical Address to User Virtual Address

Purpose:

In real mode, the **PhysToUVirt** function converts a 32-bit physical address to a valid segment:offset pair, if the address is below one megabyte. In protected mode, the **PhysToUVirt** converts a 32-bit physical address to a valid selector:offset pair that can be addressed from the current LDT.

Calling Sequence:

```
MOV  AX,AddressHigh          ;32-bit physical address
MOV  BX,AddressLow           ; or selector if request type 2
MOV  CX,Length               ;Length of area
MOV  DH,RequestType          ;Type of request
MOV  DL,DEVHLP_PHYSTOUVIRT
CALL [Device_Help]
```

where:

AddressHigh and *AddressLow* are the high and low bits of the 32-bit physical address, or the selector if request type is 2.

Length is the length, in bytes, of the area. Its value less than or equal to 65,535 where zero means 65,536 bytes.

RequestType specifies the type of request with one of the following:

Value	Meaning
0	Get virtual address, make segment readable/executable
1	Get virtual address, make segment readable/writable
2	Free virtual address (protected-mode-only)

Returns:

C —set if error.

AX = Error code:

- Invalid address.

C —clear if successful.

ES:BX = Segment/selector:offset pair (for request types 0 and 1)

- Valid selector:offset pair.

Remarks:

The **PhysToUVirt** function is typically used to provide a caller of a device driver with addressability to a fixed memory area, such as ROM code and data. The device driver must know the physical address of the memory area to be addressed.

The **PhysToUVirt** function leaves its result in ES:BX.

In protected mode, the segment:offset pair returned in the BX register is zero for request types 0 and 1. But for request type 2, the AX register contains a selector on entry to **PhysToUVirt**, and the BX and CX registers are ignored.

Note

The **PhysToUVirt** function can also be used in protected mode to free a selector returned on a prior **PhysToUVirt** call.

Note that in protected mode, the offset returned is *always* zero.

PhysToVirt: Map Physical Address to Virtual Address

Purpose:

In real mode, the **PhysToVirt** function converts a 32-bit address to a segment:offset pair. In protected mode, **PhysToVirt** converts a 32-bit address to a valid selector:offset pair.

Calling Sequence:

```
MOV  BX,AddressLow          ;32-bit physical address
MOV  AX,AddressHigh         ;
MOV  CX,Length              ;Length of segment
MOV  DH,Result              ;Leave result
MOV  DL,DEVHLP_PHYSTOVIRT
CALL [Device_Help]
```

where:

AddressLow and *AddressHigh* are the low and high words of the 32-bit physical address to be converted.

Length should be set to the length, in bytes, of the segment transfer.

Result specifies where the virtual address result of the function will be returned. It is one of the following:

Value	Meaning
0	Return result in DS:SI
1	Return result in ES:DI

Returns:

C —set if error.

AX = Error code:

- Invalid address.

C —clear if successful.

- DS:SI

  Valid virtual address if the DH register was zero.

- ES:DI

  Valid virtual address if the DH register was 1.

Z —clear if no change in addressing mode.

Z —set if addressing mode has changed

- Previously stored addresses must be recalculated.

Remarks:

PhysToVirt provides addressability to data for bimodal operations performed during task time and interrupt time. The interrupt handler of a bimodal device driver must be able to address data buffers regardless of the context of the current process. This is true because the current LDT (Local Descriptor Table) will not necessarily address the data space that contains the data buffer that the interrupt handler needs to access.

This function performs mode-dependent addressing on behalf of a device driver, relieving it of the need to recognize the CPU mode and the subsequent effects on accessing memory. However, this function is essential when the device driver needs to access a memory location at both task and interrupt time. This restriction applies because the context at interrupt time may differ from that at task time.

UnPhysToVirt is not required whenever **PhysToVirt** is used except as follows:

- When use of the converted address is ended (no more **PhysToVirt** calls)
- Before the procedure that issued **PhysToVirt** returns to its caller

In addition, multiple **PhysToVirt** calls may be performed prior to issuing the **UnPhysToVirt** call. Only one call to **UnPhysToVirt** is needed.

PhysToVirt leaves its *result* in either ES:DI or DS:SI, giving the caller the ability to move strings in either direction. Upon return, interrupts are off if the processor is in real mode and if the physical address is above one megabyte.

Typical use of **PhysToVirt** function is to convert the physical address of a buffer to a virtual address so that data may be transferred in or out at interrupt time.

PhysToVirt is guaranteed to preserve registers CS, SS, SP, and DS if called with the DH register equal to 1, or ES if called with the DH register equal to zero.

The only exception to this guarantee is when a mode switch occurs. If the system decides to switch to protected mode for an address that lies above the one-megabyte address boundary, and if the current mode is real mode, then:

- The segment addresses in the CS and SS registers are set for the current mode
- The SP register is reserved
- The DS register is set for the current mode only if it contains the data segment value of the device driver and if it is not being used for the converted address.

Note

In the event of a mode switch, any previously stored address pointers that contain the DS register for the device driver data segment must be stored again by the device driver. The zero flag (ZF) is set of a change in address mode occurred. In this case, the device driver must recalculate and store again any buffer addresses that were previously saved.

If a **PhysToVirt** call had previously been done with the address mode unchanged, and if a subsequent **PhysToVirt** requires a switch to protected mode, then the previously converted **PhysToVirt** address is considered invalid for the current mode; **PhysToVirt** must be reissued to recalculate the address.

When **PhysToVirt** is being used to recalculate an address after a mode switch occurs, it (the second **PhysToVirt**), will not cause a mode switch. The previous address is then valid and preserved (as long as the recalculation uses the opposite segment register from the one that originally caused the mode switch).

The pool of temporary selectors used by **PhysToVirt** in protected mode is not dynamically extendable. The converted addresses are valid as long as the device driver does not relinquish control via **Block**,**Yield**, or RET. An interrupt handler may use converted addresses prior to its EOI, with interrupts enabled. If an interrupt handler needs to use converted addresses after its EOI, it must protect the converted addresses by running with interrupts disabled.

The segment length parameter may be set to the length of the transfer.

Hint

For performance reasons, a device driver should try to optimize its usage of **PhysToVirt** and **UnPhysToVirt**. For the first **PhysToVirt** call that the device driver makes, it should pick the address that is likely to cause a mode switch and use the ES register. This would permit the mode switch to take place and retain the driver's data segment in the DS register.

The device driver must not enable interrupts or change the returned segment register (ES or DS) before it has finished using the returned value. The *value* returned in the segment register has no physical meaning, so the caller of the **PhysToVirt** should no have reason to examine it. While the pointer(s) generated by **PhysToVirt** are in use, the device driver may call only for another **PhysToVirt**. It may not call any other **DevHlp** routines, since they may not preserve the special ES/DS values.

On completing the operation with the virtual address, the device driver must restore the previous interrupt state. The returned virtual address will not be valid after a **Block**.

See Also:

Block, UnPhysToVirt, Yield

Unlock: Unlock Memory Segment

Purpose:

The **Unlock** function unlocks a locked memory segment in any mode.

Calling Sequence:

```
MOV  BX,LockHandleLow          ;Handle for segment
MOV  AX,LockHandleHigh         ; returned by Lock
MOV  DL,DEVHLP_UNLOCK
CALL [Device_Help]
```

where:

LockHandleLow and *LockHandleHigh* are the low and high words of the segment handle returned by **Lock**.

Returns:

C —clear if segment unlocked, set if segment locked.

See Also:

Lock

UnPhysToVirt: Mark Completion of Physical Address Use

Purpose:

The **UnPhysToVirt** call is required to mark the completion of address conversion from **PhysToVirt** calls.

Calling Sequence:

```
MOV  DL,DEVHLP_ UNPHYSTOVIRT
CALL [Device_ Help]
```

Returns:

Z —set if the address mode changed

Z —cleared if no address mode changed

Remarks:

This function forms part of the encapsulation of mode-dependent addressing on behalf of a device driver, relieving it of the need to recognize the CPU mode and the subsequent effects on accessing memory.

UnPhysToVirt must be called by the same procedure that issued **PhysToVirt** when use of converted addresses is completed and *before* the procedure returns to its caller. The procedure that had called **PhysToVirt** may call other procedures before calling **UnPhysToVirt**. Multiple **PhysToVirt** calls may be issued prior to issuing the **UnPhysToVirt** call. Only one call to **UnPhysToVirt** is needed.

The zero flag (ZF) is set if a mode switch occurred. This allows the device driver to recalculate any stored pointers that were not used in the data transfer operations with the **PhysToVirt** call.

UnPhysToVirt, in the event of a switch to real mode, resets the CS and SS registers to real mode. The SP register will be preserved. The DS register will be reset to the device driver's data segment. Note that the addresses that caused the switch into protected mode cannot be preserved

or converted for real mode. The ES register will not be preserved.

See Also:

PhysToVirt

VerifyAccess: Verify Memory Access

Purpose

The **VerifyAccess** function verifies whether the user process has the correct access rights for the memory that it passed to the device driver. If the process does not have the correct access rights, it will be terminated.

Calling Sequence:

```
MOV AX,Segment              ;Selector or segment
MOV CX,MemLength            ;Length of memory area
MOV DI,MemOffset            ;Offset to memory area
MOV DL,DEVHLP_VERIFYACCESS
CALL [Device_Help]
```

where:

Segment is the selector or segment of the memory segment to be verified for access.

MemLength is the length, in bytes, of the memory area to be verified for access. Zero means 64KB.

MemOffset is the offset to the memory segment. It has one of the following values:

Value	Meaning
0	Read access
1	Read/write access

Returns:

C —clear if no error, access verified; set if error, access attempt failed.

Remarks:

A device driver can receive an address to memory as part of a Generic IOCtl request from a process. Since the operating system cannot verify addresses embedded in the IOCtl call, the device driver must request verification to prevent itself from accidentally destroying memory for a user process. If the verification test fails, **VerifyAccess** terminates the process.

Note that verification may take place only in protected mode. If **VerifyAccess** is called in real mode, it returns that the memory is accessible.

Once **VerifyAccess** has verified that the process has the needed access to a specific address location, the device driver need not request access verification each time it yields the CPU during task-time processing of this process's request. If the process makes a new request, however, the device driver must request access verification.

Note also that, prior to requesting a **Lock** on user process-supplied addresses, the device driver must verify the user process's access to the memory by using the **VerifyAccess** call. The device driver must not yield the CPU between the **VerifyAccess** and the **Lock**; otherwise, the user process could shrink the segment before it has been locked. Once the user access has been verified, the device driver may convert the virtual address to a physical address and lock the memory. The access verification is then valid for the duration of the lock.

See Also:

Lock

VirtToPhys: Map Virtual Address to Physical Address

Purpose:

In real mode, the **VirtToPhys** function converts a segment:offset pair to a 32-bit physical address. In protected mode, **VirtToPhys** converts a selector:offset pair to a 32-bit address.

Calling Sequence:

```
LDS  SI,Address                    ;Virtual address
MOV  DL,DEVHLP_VIRTTOPHYS
CALL DGROUP:[Device_Help]          ;DS invalid from load
```

where:

Address is the virtual address in the form segment:offset for real mode, or selector:offset for protected mode.

Returns:

C —set if error, clear if no error.

AX:BX = Physical address: 32-bit number

Remarks:

Before this function is called, the virtual address should be locked using the **DevHlp** function **Lock**—if the segment is not known to already be locked.

This function is typically used to convert a virtual address supplied by a device driver client via a Generic IOCtl call, so that the memory may be accessed at interrupt time.

The explicit DGROUP override is necessary because the DS register is invalidated prior to invoking **DevHlp**.

5.9 Interrupt Handling

This section describes device helper functions for interrupt handling. These functions are listed as follows:

Function	Description
EOI	Issue End-Of-Interrupt
SetIRQ	Set Hardware Interrupt Handler
SetROMVector	Set ROM BIOS Interrupt Handler
UnSetIRQ	Remove Hardware Interrupt Handler

EOI: Issue End-Of-Interrupt

Purpose:

The **EOI** function is used to issue an End-Of-Interrupt (EOI) to the master/slave 8259 interrupt controller as appropriate to the interrupt level.

Calling Sequence:

```
MOV  AL,IRQnum              ;Interrupt level number (0–0FH)
MOV  DL,DEVHLP_EOI
CALL [Device_Help]
```

where:

IRQnum is the interrupt level number in the range from 00H to 0FH. Any software interrupt vector may be set. Invalid ranges are 08H–0FH, 50H–57H, and 70H–77H.

Returns:

None.

Remarks:

The **EOI** function is used to issue an End-Of-Interrupt to the 8259 interrupt controller on behalf of a device driver interrupt handler. If the specified interrupt level is for the slave 8259 interrupt controller, then this routine issues the EOI to both the master and slave 8259 controllers.

Device drivers must use this service in their interrupt handlers for compatibility with future versions of MS OS/2.

This function is bimodal and may be called at initialization time for interrupt processing.

EOI does not change the state of the interrupt flag.

SetIRQ: Set Hardware Interrupt Handler

Purpose:

The **SetIRQ** function registers a device interrupt handler for a hardware interrupt level.

Calling Sequence:

```
MOV  AX,Handler          ;Interrupt handler offset
MOV  BX,IRQnum           ;Interrupt level number (0–0FH)
MOV  DH,SharedInt        ;Interrupt sharing
MOV  DL,DEVHLP_SETIRQ
CALL [Device_Help]
```

where:

Handler is the offset to the interrupt handler.

IRQnum is the interrupt level number in the range from 00H to 0FH. Any software interrupt vector may be set. Invalid ranges are 08H–0FH, 50H–57H, and 70H–77H.

SharedInt specifies whether interrupt sharing is available with one of the following values:

Value	Meaning
0	Not shared
1	Shared

Returns:

C —clear if no error, set if error.

AX = Error code:

- IRQ is not available

Remarks:

The attempt to register an interrupt handler for an IRQ that is shared will fail if the IRQ is:

- Already owned by another device driver as "not shared"
- Owned by a real-mode box interrupt handler
- The IRQ used to cascade the slave 8259 interrupt controller.

The attempt to register an interrupt handler for an IRQ that is *not* shared will fail if the IRQ is:

- Already owned by another device driver as shared or not shared
- Owned by a real-mode box interrupt handler
- The IRQ used to cascade the slave 8259 interrupt controller.

SetIRQ enables the interrupt level at the 8259 interrupt controller on behalf of the device driver interrupt handler if the IRQ is available.

The DS register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the DS register, it should restore DS to its original value.

See Also:

PhysToVirt

SetROMVector: Set ROM BIOS Interrupt Handler

Purpose:

The **SetROMVector** function replaces a real-mode software interrupt handler with a handler from the device driver; it then returns a real-mode pointer to the previous interrupt handler for chaining.

Calling Sequence:

```
MOV  AX,Handler                ;Interrupt handler offset
MOV  BX,IRQnum                 ;Interrupt number
MOV  SI,SaveDS                 ;Offset to save real-mode DS
MOV  DL,DEVHLP_SETROMVECTOR
CALL [Device_Help]
```

where:

Handler is the offset to the new interrupt handler.

IRQnum is the interrupt level number in the range from 00H to 0FH. Any software interrupt vector may be set. Invalid ranges are 08H–0FH, 50H–57H, and 70H–77H.

SaveDS is the offset (from the CS register) to save the real-mode DS.

Returns:

C —clear if no error.

AX:DX = Real-mode pointer to previous handler.

C —set if error.

AX = Error code:

- Invalid interrupt number.

Remarks:

SetROMVector allows a bimodal device driver to hook a real-mode ROM BIOS interrupt and perform any interlocking that may be necessary to coordinate device I/O between the driver and ROM BIOS.

Any software interrupt vector may be set. Invalid ranges are 8–0FH, 50–57H, and 70–77H.

The device driver's interrupt handler for the real-mode software interrupt receives control directly on ocurrence of the interrupt. Consequently, the DS register is not set up for the handler on entry. Instead, the handler must set the DS register with the value that **SetROMVector** had previously saved. Note that the location in which the real-mode value for DS is saved must be in the handler's code segment (CS); otherwise, the handler will be unable to set up DS.

The DS register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the DS register, it should restore DS to its original value.

See Also:

PhysToVirt

UnSetIRQ: Remove Hardware Interrupt Handler

Purpose:

The **UnSetIRQ** function removes the current hardware interrupt handler.

Calling Sequence:

```
MOV  BX,IRQnum              ; Interrupt level number (0–0FH)
MOV  DL,DEVHLP_UNSETIRQ
CALL [Device_Help]
```

where:

IRQnum is the interrupt level number, in the range from 00H to 0FH previously set by the **SetIRQ** call.

Returns:

None.

Remarks:

None.

5.10 Timer Services

This section describes the device helper functions for managing the timer handler. Following is a list of these functions:

Function	Description
ResetTimer	Reset Timer Handler
SetTimer	Set Timer Handler
TickCount	Modify Timer

ResetTimer: Reset Timer Handler

Purpose:

The **ResetTimer** function removes a timer handler for the device driver.

Calling Sequence:

```
MOV  AX, TimerHandler        ;Offset to timer handler
MOV  DL,DEVHLP_RESETTIMER
CALL [Device_Help]
```

where:

TimerHandler is the offset to the timer handler to be removed from the timer handler list.

Returns:

C —cleared if no error, set if error.

AX = Error code:

- Address not found.

Remarks:

Timer handlers are analogous to the user timer interrupt (INT 1CH). For information about timer handlers, see the **SetTimer** function.

The DS register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the DS register, it should restore DS to its original value.

See Also:

PhysToVirt, SetTimer

SetTimer: Set Timer Handler

Purpose:

The **SetTimer** function adds a timer handler to the list of timer handlers to be called on a timer tick.

Calling Sequence:

```
MOV  AX, TimerHandler          ;Offset of timer handler.
MOV  DL,DEVHLP_SETTIMER
CALL [Device_Help]
```

where:

TimerHandler is the offset of the timer handler to be added to the list of timer handlers.

Returns:

C —cleared if no error, set if error.

AX = Error code:

- Timer handler disallowed (maximum number of handlers reached or timer handler already set).

Remarks:

A driver driver may use a timer handler to drive a non-interrupt device instead of using time-outs with the **Block** and **Run** services. This method is preferable because when measured on a character-by-character basis, **Block** and **Run** are costly, requiring one or more task switches per character I/O.

While a timer handler is in the format of a CALL/RETURN routine (when it is finished processing, it performs a normal return), it operates in interrupt mode. The timer handler is analogous to the user timer (INT 1CH) handler.

Note

Be sure to remain in the handler for as short a period of a time as possible.

Note that the **SetTimer** function returns an error when it is called with a *TimerHandler* address that is already on its list.

The DS register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the DS register, it should restore DS to its original value.

See Also:

Block, PhysToVirt, Run

TickCount: Modify Timer

Purpose:

The **TickCount** function registers a new timer handler or modifies a previously registered timer handler to be called on every *n* timer ticks instead of every timer tick.

Calling Sequence:

```
MOV  AX, TimerHandler        ;Offset to timer handler
MOV  BX, Count               ;Number of tick counts (0-0FFFFH)
MOV  DL,DEVHLP_TICKCOUNT
CALL [Device_Help]
```

where:

TimerHandler is the offset to the timer handler for which the tick count is to be modified.

Count is the number of tick counts in the range from 00H to 0FFFFH where zero means 0FFFFH+1 ticks.

Returns:

C —cleared if no error, set if error.

AX = Error code:

- Timer handler cannot be modified or set.

Remarks:

For a new timer handler, **TickCount** registers the timer handler to be called every *Count* timer ticks instead of every timer tick.

For a previously-registered timer handler, **TickCount** changes the number of ticks that must take place before the timer handler gets control. This allows device drivers to support the time-out function without needing to count timer ticks.

At task-time, this **DevHlp** may be used to modify a timerhandler registered via **SetTimer** or it may be used to register a new timer handler that is initially invoked every *Count* ticks.

In user mode (during task-time) or interrupt mode (during interrupt-time), this **DevHlp** may only be used to modify a previously-registered timer handler. This allows an interrupt handler to reset the timing condition at interrupt-time.

Note that **SetTimer** sets a default *Count* of 1. Multiple **TickCount** requests may be issued for a given timer handler, but only the last **TickCount** setting will be in effect.

TickCount affects only the specified registered timer handler. It has no effect on other timer handlers.

An error is returned via the carry flag (CF) if the timer handler cannot be modified or set.

The DS register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the DS register, it should restore DS to its original value.

See Also:

PhysToVirt, SetTimer

5.11 Monitor Management

This section describes the device helper functions for managing monitors. Following is a list of these functions:

Function	Description
DeRegister	Remove Monitor
MonFlush	Flush Data from Monitor Stream
MonitorCreate	Create Monitor
MonWrite	Pass Data Records to Monitors
Register	Add Monitor

DeRegister: Remove Monitor

Purpose:

The **DeRegister** function removes the monitor associated with the specified task from the monitor chains.

Calling Sequence:

```
MOV  BX,MonitorPID          ; Process ID of monitor task
MOV  AX,MonitorHandle       ; MonitorCreate handle for chain
MOV  DL,DEVHLP_DEREGISTER
CALL [Device_Help]
```

where:

The *MonitorPID* and *MonitorHandle* parameters are supplied by the device driver.

Returns:

C —clear if no error, set if error.

AX = Error code:

- Invalid monitor handle.

Remarks:

This function may be called only at task time.

See Also:

Register

MonFlush: Flush Data from Monitor Stream

Purpose:

The **MonFlush** function removes all data from the monitor stream.

Calling Sequence:

```
MOV  AX,MonitorHandle        ; MonitorCreate handle for chain
MOV  DL,DEVHLP_MONFLUSH
CALL [Device_Help]
```

where:

MonitorHandle is the **MonitorCreate** handle for the chain.

Returns:

C —clear if no error, set if error.

AX = Error code:

- Invalid monitor handle.

Remarks:

This function may be called only at task time.

Until the flush completes, subsequent **MonWrite** requests will fail (or block).

The general format of monitor records requires that every record contain a flag word as the first entry. One of the flags indicates that this record is a flush record, which consists of the flag word only. Monitors use this flush record along the chain to reset internal state information and to ensure that all internal buffers are flushed. The flush record must be passed along to the next monitor, since the monitor dispatcher will not process any more information until the flush record is received at the end of the chain.

See Also:

MonitorCreate, MonWrite

MonitorCreate: Create Monitor

Purpose:

The **MonitorCreate** function creates an initially empty chain of monitors.

Calling Sequence:

```
LES  SI,FinalBuffer          ; Address of final buffer
LDS  DI,NotifyRtn            ; Address of notification routine
MOV  AX,Handle               ; Handle for this chain
MOV  DL,DEVHLP_MONCREATE
CALL [Device_Help]
```

where:

FinalBuffer specifies the final buffer for this monitor chain.

NotifyRtn specifies the address of the notification routine.

Handle = 0 means create a new monitor chain.

Handle ≠ 0 means that the handle returned from a previous **MonitorCreate** call indicated that this chain is to be removed. All monitor tasks registered with this chain must be deregistered before this call is made.

Returns:

C —clear if no error.

AX = monitor chain handle if *Handle* was zero.

C —set if error.

AX = Error code:

- Invalid monitor handle.

- Not enough memory.

Remarks:

A monitor chain is a list of monitors, with a device driver buffer address and code address as the last element on this list. Data is placed in a monitor chain by the **MonWrite** function. The monitor package then feeds the data through all registered monitors, putting the resulting data, if any, into the device driver's specified buffer. When data are placed in this buffer, the device driver's notification routine is called at task time. The device driver should initiate any necessary action in a timely fashion and return from the notification entry point without delay.

The **MonitorCreate** function establishes one of these monitor chains. The chains are created empty so that data written into them is placed immediately into the output buffer.

This routine can also destroy a monitor chain if the *Handle* parameter (AX) is nonzero. The nonzero value is the handle of the chain to remove.

Note

If the **MonWrite** function is called at interrupt time and if the monitor chain is empty, the driver notification routine will be called at interrupt time. Under all other circumstances it is called at task time.

Notification Routine Considerations:

The notification routine *NotifyRtn* is called by the monitor dispatcher when the final buffer has been filled in. The monitor dispatcher calls the specified address and holds the final buffer address in ES:SI. The DS register is set to the correct value for the device driver.

The device driver must process the contents of the *FinalBuffer* before returning to the monitor dispatcher. This entry point will be called only in protected mode.

See Also:

MonWrite

MonWrite: Pass Data Records to Monitors

Purpose:

The **MonWrite** function passes data records to the monitors for filtering.

Calling Sequence:

```
LDS  SI,DataRecord          ; Address of data record
MOV  CX,Count               ; Byte count of data record
MOV  AX,MonitorHandle       ; MonitorCreate handle for chain
MOV  DH,WaitFlag            ; Wait/No Wait flag
MOV  DL,DEVHLP_MONWRITE
CALL [Device_Help]
```

where:

All the parameters are supplied by the device driver.

The *WaitFlag* is zero if the **MonWrite** request occurs at task or user time and if the device driver wishes to have the monitor dispatcher perform the synchronization. If no wait is required, a value of one is specified. At interrupt time, it is necessary to specify a value of one.

Returns:

C —clear if no error, and set if error.

AX = Error code:

- Invalid monitor handle.
- Not enough memory.

Remarks:

The **MonWrite** function may be called at either task or interrupt time. The "Not enough memory" condition can arise when the **MonWrite** call is made and the buffer does not contain sufficient free space to receive the data. If this condition occurs at interrupt time, it signifies an overrun. If it occurs at task (or user) time, the process can block.

A **MonFlush** call that is in progress can also cause a "not enough memory" condition. Waiting until the **MonFlush** call has completed may correct this condition.

See Also:

MonFlush, MonitorCreate

Register: Add Monitor

Purpose:

The **Register** function adds a monitor to the chain of monitors for a class of device.

Calling Sequence:

```
LES SI,InputBuffer              ; Address of input buffer
MOV DI,OutputBufferOffset       ; Offset of output buffer
MOV CX,MonitorPID               ; Process ID of monitor task
MOV AX,MonitorHandle            ; MonitorCreate handle for chain
MOV DH,PlacementFlag            ; High or low place in chain
MOV DL,DEVHLP_REGISTER
CALL [Device_Help]
```

where:

The *InputBuffer*, *OutputBufferOffset*, and *PlacementFlag* parameters are supplied by the monitor task request.

The *MonitorPID* and *MonitorHandle* parameters are supplied by the device driver.

Returns:

C —clear if no error, set if error.

AX = Error code:

- Invalid monitor handle.
- Not enough memory.
- Monitor buffer too small.

Remarks:

This function may be called only at task time.

See Also:

DeRegister

5.12 System Services

This section describes the system services device helper functions, those functions used for general servicing of the MS OS/2 operating system. Following is a list of these functions:

Function	Description
GetDosVar	Return Pointer to DOS Variable
GrantPortAccess	Grant Access to I/O Ports
PortUsage	Indicate I/O Port Usage
ROMCritSection	Flag Critical Section of Execution
SendEvent	Indicate Event

GetDosVar: Return Pointer to DOS Variable

Purpose:

The **GetDosVar** function returns the address of an internal DOS variable.

Calling Sequence:

```
MOV  AL, VarNumber          ;Variable wanted
MOV  DL,DEVHLP_GETDOSVAR
CALL [Device_Help]
```

where:

VarNumber is the number of the MS OS/2 variable to be returned.

Returns:

C —cleared if no error.

AX:BX points to the variable.

C —set if error.

Remarks:

The list of available variables is subject to growth in future versions of the operating system. The operating system maintains these variables for the benefit of device drivers.

The returned pointer (address) is a bimodal pointer. Note that the address returned is that of the indicated variable, which may contain a structure or a vector to a facility.

The variables are read-only and are described as follows:

Index	Value	Comment
1	*SysINFOSeg*:WORD	Bimodal segment address of the system (GDT) INFO segment. Valid at both task time and interrupt time.
2	*LocINFOSeg*:DWORD	Selector/Segment address of the local (LDT) INFO segment. Valid only at task time.
3	*Com1PID*:WORD *Com2PID*:WORD	Owners of *COM1* and *COM2* Valid at both task time and interrupt time.
4	*VectorS*:DWORD	Reserved
5	*VectorReboot*:DWORD	Vector to reboot the DOS. Valid at both task time and interrupt time.
6	*VectorM*:DWORD	Reserved
7	*YieldFlag*:BYTE	Indicator for performing yields. Valid only at task time.
8	*TCYieldFlag*:BYTE	Indicator for performing time critical yields. Valid only at task time.

See Also:

TCYield, Yield

SendEvent: Indicate Event

Purpose:

The **SendEvent** function is called by a device driver to indicate the occurrence of an event.

Calling Sequence:

```
MOV  AH,Event                ;Event being signaled
MOV  BX,Argument             ;Parameter for event being signaled
MOV  DL,DEVHLP_SENDEVENT
CALL [Device_Help]
```

where:

Event identifies the event being signaled with one of the following values:

Value	Meaning
0	Session manager hot key from the mouse device *Argument* is a two-byte time stamp where the high byte is seconds and the low byte is hundredths of seconds.
1	CONTROL-BREAK *Argument* is reserved and must be zero.
2	CONTROL-C *Argument* is reserved and must be zero.
3	CONTROL-SCROLL LOCK *Argument* is reserved and must be zero.
4	CONTROL-PRINTSCREEN *Argument* is reserved and must be zero.
5	SHIFT-PRINTSCREEN *Argument* is reserved and must be zero.

6	Session manager hot key from the keyboard device *Argument* is the *Hot key ID* defined via the keyboard IOCtl, **Function 56H**, Set Session Manager Hot Key.

Argument is the parameter for the event being signaled.

Returns:

C —set if error, and cleared if no error.

Remarks:

None.

ROMCritSection: Flag Critical Section of Execution

Purpose:

The **ROMCritSection** function, which is called by the portion of a device driver that intercepts a ROM BIOS software interrupt (a ROM BIOS compatibility handler), flags a critical section of execution in the ROM BIOS to prevent the real-mode session from being frozen.

Calling Sequence:

```
MOV  AL,EnterOrExit          ;Critical section flag:
MOV  DL,DEVHLP_ROMCRITSECTION
CALL [Device_Help]
```

where:

EnterOrExit is a critical-section flag with one of the following values:

Value	Meaning
0	Exit critical section
1	Enter critical section

Returns:

None.

Remarks:

Sections of ROM BIOS code must be protected from preemption that occurs when a user switches away from the real-mode session, causing it to be suspended in the background. Some I/O processing, however, cannot tolerate being suspended. Specific examples are the printer (BIOS INT 17H), disk (BIOS INT 13H), and screen (BIOS INT 10H). It is the responsibility of the MS OS/2 device driver to intercept the appropriate ROM BIOS interrupt and issue the **DevHlp** function **ROMCritSection** to protect the ROM BIOS critical section of execution.

Warning

When the MS OS/2 device driver issues **ROMCritSection** to "enter" a ROM BIOS critical section, you will not be able to switch away from the screen of the real-mode session to another screen. This may cause you problems. For example, if a real-mode terminate-and-stay-resident program takes control while the CPU is executing the ROM BIOS, the time spent in the ROM BIOS critical section will be longer, and you will be unable to switch screens.

The worst case occurs when the terminate-and-stay-resident program is interactive, not allowing the MS OS/2 device driver to issue the "exit" from the critical section and not allowing you to switch away from the real-mode session screen until you terminate the program.

PortUsage: Indicate I/O Port Usage

Purpose

The **PortUsage** function indicates the usage of I/O ports by the device driver.

Calling Sequence:

```
MOV  AX,FirstPort          ;First port
MOV  BX,LastPort           ;Last port
MOV  CL,UseIndicator       ;Usage
MOV  DH,TypeOfAccess       ;Request or Release
MOV  DL,DEVHLP_PORTUSAGE
CALL [Device_Help]
```

where:

FirstPort is either the starting port of a contiguous range, or a single port.

LastPort is either the ending port of a range or equal to the value in *FirstPort*.

UseIndicator informs the operating system of how the device driver uses the ports. It has one of the following values:

Value	Meaning
0	Non-exclusive (common) use
1	Exclusive-use-only

If the ports are exclusive to the device, the device driver indicates them as "exclusive-use-only." If the ports are "non-exclusive" to the device—that is, if they are shared among several devices or device drivers—then the device driver indicates the ports as common.

TypeOfAccess indicates whether the ports are being claimed (reserved) or released. It has one of the following values:

Value	Meaning
0	Request access to I/O port
1	Release access to I/O port

Returns:

C —clear if no error, set if error.

AX = Error code:

- Access denied.

Remarks

The attempt to reserve the I/O port(s) may fail under the following circumstances:

- Registration for a port as exclusive-use-only will fail if the port has been previously marked as either common or for exclusive use (that is, it will fail if the port has already been reserved by another device driver or by an application IOPL segment).
- Registration for a port as common will fail if the port has been previously marked for exclusive use.

The function **PortUsage** is used for reservation of I/O ports for a device driver. These ports are for program I/O, for querying or setting the state of the device, and for other such operations. This includes ports used for individual DMA channels. A device driver that does not register its port usage runs the risk of another device driver registering for the same ports.

A device driver that uses **PortUsage** will be notified via the device request packet Port Access (Command Code 21) when an IOPL application requests port access to ports that the device driver has registered as exclusive-use-only. The device driver may choose to allow the IOPL application access to the ports by issuing the **GrantPortAccess** function call. By using this function, a device driver can prevent another device driver or IOPL application from using its ports.

See Also:

GrantPortAccess

GrantPortAccess: Grant Access to I/O Ports

Purpose

The **GrantPortAccess** function grants an IOPL application access to I/O ports that had been registered by the device driver for exclusive-use-only (see also **PortUsage** in this section). This function also sets the per-process bitmap of ports that the application is allowed to access.

Calling Sequence:

```
MOV  AX,FirstPort          ;First port
MOV  BX,LastPort           ;Last port
MOV  DH,TypeOfAccess       ;Request or Release
MOV  DL,DEVHLP_GRANTPORTACCESS
CALL [Device_Help]
```

where:

FirstPort is either the starting port of a contiguous range, or a single port.

LastPort is either the ending port of a range, or it is equal to *FirstPort*.

TypeOfAccess indicates whether the ports being claimed (reserved) or released. It has one of the following values:

Value	Meaning
0	Request access to I/O port
1	Release access to I/O port

Returns:

C —clear if no error, set if error.

- Access denied (ports already reserved).

Remarks

For IOPL applications that need to do IN/OUT instructions in an IOPL segment, the **GrantPortAccess** call can be used to grant the process access to the I/O ports.

The attempt by the device driver to grant access to the I/O port(s) may fail if the port(s) is not registered by the device driver with the **PortUsage** call.

See Also:

PortUsage

Appendix A
Translate Table Format

This appendix describes the format and contents of the *Translate Table*, which MS OS/2 uses to translate new keystrokes. The table includes 127 copies of the KeyDef record (one for each possible scan code that may be returned from the keyboard).

Not all the entries are used; those that aren't used are zero. The entries are in scan code order, based on the remapped scan codes returned by the keyboard controller when it is in compatibility mode. Compatibility mode translates keyboard scan codes to scan codes based on the position of the keys as they were on the standard PC keyboard (plus additional keys on the Enhanced Keyboard). It also converts key "break" codes to the equivalent scan code with the high-order bit turned on (that is, it adds 128 to the key code).

Translate Table Format

The *Translate Table* has the following format:

Name	**Parameter**
XHeader	*XHeader*
KeyDef1	*KeyDef*
KeyDef2	*KeyDef*
⋮	
KeyDef127	*KeyDef*
CtlAltTbl	*CtlAltKeys*
AccentTbl	*AccentTable*

Translate Table Parameters

This section describes each of the parameters in the *Translate Table.*

The XHeader Parameter

The *XHeader* parameter in the translate table has the following format:

Length	Parameter
WORD	*XTableID* Reserved
WORD	*XTableFlags1* This flag word has the following bitmap:

Bit	Meaning
0	Reserved = 0
1	*AltGrafL* Use left ALT key as ALT-Graphics
2	*AltGrafR* Use right ALT key as ALT-Graphics
3	*ShiftLock* Treat CAPSLOCK as SHIFTLOCK
4–15	*Reserved*

WORD	*XTableFlags2* Reserved = 0
WORD	*KbdType* Reserved = 0
WORD	*KbdSubType* Reserved = 0

WORD	*XTableLen* Length of translate table (in bytes)
WORD	*EntryCount* Number of *KeyDef* entries
WORD	*EntryWidth* Width of *KeyDef* entries (in bytes)
WORD	*Country* Reserved
11 WORDS	*Reserved* Reserved = 0

The KeyDef Parameter

There are 127 copies of the *KeyDef* parameter in the translate table. Each copy has the following format:

Length	Parameter
BYTE	*Char1* Use depends on *KeyType*
BYTE	*Char2* Use depends on *KeyType*
BYTE	*Char3* Use depends on *KeyType*
BYTE	*Char4* Use depends on *KeyType*
WORD	*XlateOp* Translate-operation specifier with the following bitmap:

Bit	Meaning
0–6	*AccentFlags* See "Note 1: Accent Flags" and "Note 9: Accent Key"
7–10	*CtlAltKey* See "Note 2: Control-Alt Keys"
11–15	*KeyType* See "Note 3: Key Types"

The CtlAltKeys Parameter

The *CtlAltKeys* parameter in the translate table is a table of CONTROL-ALT key translations. It has the following format:

Length	Parameter
BYTE	*Pad* Zero byte to keep things on word boundaries
BYTE	*CtlAltKey1* See "Note 2: Control-Alt Keys"
BYTE	*CtlAltKey2*
⋮	
BYTE	*CtlAltKey15*

The AccentTable Parameter

The *AccentTable* parameter in the translate table is a table containing seven *AccentEntry*, accent key definitions. The *AccentTable* has the following format:

Name	Parameter
AccentEntry1	*AccentEntry*
AccentEntry2	*AccentEntry*
:	
:	
AccentEntry7	*AccentEntry*

Each *AccentEntry* in the *AccentTable* is defined as follows (See "Note 1: Accent Flags" and "Note 9: Accent Key"):

Length	Parameter
2 BYTES	*NonAccent* Character/scan code for key when not used as accent
2 BYTES	*CtlAccent* Character/scan code for key when used with CONTROL key
2 BYTES	*AltAccent* Character/scan code for key when used with ALT key
2 BYTES	*Map1* From/to character pair for accented translation
2 BYTES	*Map2* From/to character pair for accented translation
	:
	:
2 BYTES	*Map20* From/to character pair for accented translation

Notes about the Translation Table

Note 1: Accent Flags

The *AccentFlags* field of the *KeyDef* record has seven flags that are individually set if a corresponding entry in the *Accent Table* applies to this scan code. If the key pressed immediately before the current one was an accent key, and if the bit for that accent is set in the *AccentFlags* field for the current key, then the corresponding *AccentTable* entry is searched for the replacement character value to use. If no replacement is found, the "not-an-accent" beep sounds and the accent character and current character are passed as two separate characters. See also "Note 9: Accent Keys."

Note 2: Control-Alt Keys

The *CtlAltKey* field is an offset into the *CtlAltKeys* table. It is zero if this key does not have a special translation for the corresponding scan code when the CONTROL and ALT shift keys are pressed along with it. Otherwise, the character code to use when that key is pressed must be in the *CtlAltKeys* table.

Note

This means only 15 keys can be defined for special CONTROL-ALT characters.

Note 3: Key Types

The *KeyType* field of the *KeyDef* record currently has the following values defined. The remaining values up to 1FH are undefined. In the following table, the effect of each type of shift is defined. Except where otherwise noted, when no shifts are active, *Char1* is the translated character. References to Undefined are clarified in "Note 4: Undefined Character."

Note

Either the ALT, ALTGRAF, or both may be present on a keyboard based on the *AltGrafL* and *AltGrafR* bits in the *XTableFlags1* flagword in the table header.

KeyType can have one of the following values where each value may be affected by the type of shift that occurs with that key type:

Value	Meaning
01H	*AlphaKey* Alphabetical character key

Type of Shift	Effect of Shift
SHIFT	Uses *Char2* (If CAPSLOCK, uses *Char1* value)
CAPSLOCK	Uses *Char2* (If SHIFT, uses *Char1* value)
CONTROL	Set standard control key code for this key's *Char1* value (See "Note 5: The Control Key")
ALT	Standard extended code (see "Note 8: Alt Keys")
ALTGRAF	Uses *Char3*, if it is not zero

02H *SpecKey*

Special non-alphabetical character key (no CAPSLOCK or ALT keys defined):

Type of Shift	Effect of Shift
SHIFT	Uses *Char2*
CAPSLOCK	No effect, only depends on SHIFT or CONTROL
CONTROL	See "Note 5: The CONTROL Key"
ALT	Marked Undefined

ALTGRAF	Uses *Char3* if it is not zero

03H *SpecKeyC*

Special non-alphabetical character key with CAPSLOCK (no ALT keys defined):

Type of Shift	Effect of Shift
SHIFT	Uses *Char2* (If CAPSLOCK, uses *Char1*)
CAPSLOCK	Uses *Char2* (If SHIFT, uses *Char1*)
CONTROL	See "Note 5: The CONTROL Key"
ALT	Marked Undefined
ALTGRAF	Uses *Char3* if it is not zero

04H *SpecKeyA*

Special non-alphabetical character key, with ALT (no CAPSLOCK keys defined):

Type of Shift	Effect of Shift
SHIFT	Uses *Char2*
CAPSLOCK	No effect, only depends on SHIFT, CONTROL, or ALT
CONTROL	See "Note 5: The CONTROL Key," and "Note 11: The SPACEBAR"
ALT	See "Note 8: ALT Keys"
ALTGRAF	Uses *Char3* if it is not zero

05H *SpecKeyCA*

Special non-alphabetical character key (with CAPSLOCK and ALT keys defined):

Type of Shift	Effect of Shift
SHIFT	Uses *Char2* (If CAPSLOCK, uses *Char1*)
CAPSLOCK	Uses *Char2* (If SHIFT, uses *Char1*)

CTL	See "Note 5: The CONTROL Key"
ALT	See "Note 8: ALT Keys"
ALTGRAF	Uses *Char3* if it is not zero

06H *FuncKey*

Function keys (*Char1* = *n* in F*n*, *Char2* is ignored, sets extended codes 58+*Char1* if no shift, or if F11 or F12, uses 139 and 140):

Type of Shift	Effect of Shift
SHIFT	Sets extended codes 83+*Char1*; F11 and F12 use 141 and 142, respectively
CAPSLOCK	No effect on function keys
CONTROL	Sets extended codes 93+*Char1*; F11 and F12 use 143 and 144, respectively
ALT	Sets extended codes 103+*Char1*; F11 and F12 use 145 and 146, respectively
ALTGRAF	Uses *Char3* if it is not zero

07H *PadKey*

Keypad keys (see "Note 6: The Pad Key" for a definition of *Char1*, and note that non-shifted use of these keys is fixed to the extended codes):

Type of Shift	Effect of Shift
SHIFT	Uses *Char2* (Unless NUMLOCK, see "Note 6: The Pad Key")
CAPSLOCK	No effect on pad keys (NUMLOCK does, see "Note 6: The Pad Key")
CONTROL	Sets extended codes (see "Note 6: The Pad Key")
ALT	Used to build a character (see "Note 6: The Pad Key")
ALTGRAF	Uses *Char3* if it is not zero

08H *SpecCtlKey*

Action keys that perform special actions when the CONTROL key is pressed:

Type of Shift	Effect of Shift
SHIFT	No effect on these keys
CAPSLOCK	No effect on these keys
CONTROL	Uses *Char2*
ALT	Marked Undefined
ALTGRAF	Uses *Char3* if it is not zero

09H *PrtScr*

PRINTSCREEN key (sets *Char1* normally):

Type of Shift	Effect of Shift
SHIFT	Signals the *PrtScr* function
CAPSLOCK	No effect on this key
CONTROL	Sets extended code and signals the Print Echo function
ALT	Marked as Undefined
ALTGRAF	Uses *Char3* if it is not zero

10H *CapsLock*

CAPSLOCK key. Behaves like a toggle key when the keyboard is a CAPSLOCK keyboard. When the ShiftLock bit is set in *XTableFlags1*, this key is processed as follows: any time the key is pressed, the bit in *Char1* is set in the lower byte of the system shift status word and left on until the left or right SHIFT key is pressed, at which time the bit is cleared.

0AH *SysReq*

SYSREQ key; treated like a SHIFT key (See "Note 7: State Keys")

0BH *AccentKey*

Keys that affect the "next" key pressed (also known as dead

keys). *Char1* is an index into the *AccentTbl* field of the *Translate Table*, selecting the *AccentEntry* that corresponds to this key. *Char2* and *Char3* do the same for the shifted *Accent* character.

Type of Shift	Effect of Shift
SHIFT	Use *Char2* to index to applicable *AccentEntry*
CAPSLOCK	No effect on this key
CONTROL	Use *CtlAccent* character from *AccentEntry* (see "Note 9: Accent Key")
ALT	Use *AltAccent* character from *AccentEntry* (see "Note 9: Accent Key")
ALTGRAF	Use *Char3* to index to applicable *AccentEntry*

Note

Each of the following state key entries set *Char1* and *Char2* to mask values as defined in "Note 7: State Keys."

0CH *ShiftKeys*

SHIFT or CONTROL key, sets/clears flags. *Char1* holds the bits in the lower byte of the shift status word to set when the key is down and clear when the key is released. *Char2* does the same thing for the upper byte of the shift status word, unless the "secondary" key prefix (E0H) is seen immediately prior to this key, in which case *Char3* is used in place of *Char2*.

0DH *ToggleKey*

General toggle key (like CAPSLOCK). *Char1* holds the bits in the lower byte of the shift status word to toggle on the first make of the key after it is pressed. *Char2* holds the bits in the upper byte of the shift status word to set when the key is down and clear when the key is released, unless the "secondary" key prefix (E0H) is seen immediately prior to this key, in which case *Char3* is used in place of *Char2*.

0EH *AltKey*

ALT key. Treated just like *ShiftKeys* above, but has its own key type because when seen, the accumulator used for *AltPadkey* entry is zeroed to prepare such entry. See "Note 6: The Pad Key" for more information about *AltPadKey* entry. Sometimes this key is treated as *AltGraphics* key if one of the *AltGraf* bits is on in *XTableFlags1*.

0FH *NumLock*

NUMLOCK key. Behaves like *ToggleKey* normally, but the keyboard device driver will set a pause screen indication when this key is pressed in combination with the CONTROL key. The pause is cleared on the next keystroke, if that keystroke generates a character.

10H *CapsLock*

CAPSLOCK key. This key is treated like a type 0DH toggle key. It has a separate entry here so that if it can be processed like a SHIFTLOCK key when that flag is set in the *XTableFlags1* word in the header. When treated as a *ShiftLock*, the *CapsLock* flag in the shift status word is set ON when this key is pressed, and only cleared when the left or right SHIFT key is depressed. *Char2* and *Char3* are processed the same as *ToggleKey*.

11H *ScrollLock*

SCROLL LOCK key. Behaves like *ToggleKey* normally, but has a separate entry here so that when used with CONTROL it can be recognized as CONTROL-BREAK.

12H *XShiftKey*

Extended SHIFT key (for National Language Support). See "Note 10: Extended State Keys" for more information.

13H *XToggleKey*

Extended Toggle key (for National Language Support). See "Note 10: Extended State Keys" for more information.

Note 4: Undefined Character

Any key combination that does not fall into any of the defined categories (for example, the CONTROL key pressed with a key that has no defined control mapping) will be mapped to the value zero, and the keytype will be set in the *KeyPacket* record indicating Undefined translation. The *KeyPacket* record passed to the monitors (if any are installed) will contain the original scan code in the *ScanCode* field and the zero in the *Char* field for this key. No *CharData* records with an undefined character code will be placed in the keyboard input buffer.

Note 5: The CONTROL Key

There are six possible situations for when another key is pressed with the CONTROL key:

Situation 1

The key pressed is an *AlphaKey* character. In this case, the CONTROL-*Char1* combination defines one of the standard defined control codes (all numbers being decimal):

CONTROL	Mapping	Codename
a	1	SOH
b	2	STX
c	3	ETX
d	4	EOT
e	5	ENQ
f	6	ACK
g	7	BEL
h	8	BS
i	9	HT
j	10	LF
k	11	VT
l	12	FF
m	13	CR
n	14	SO
o	15	SI
p	16	DLE
q	17	DC1
r	18	DC2
s	19	DC3

t	20	DC4
u	21	NAK
v	22	SYN
w	23	ETB
x	24	CAN
y	25	EM
z	26	SUB

Note

Any key defined as *AlphaKey* will use the *Char1* code value minus 96 (ASCII code for *a*) plus 1 to set the mapping shown above. So any scan code defined as *AlphaKey* must assign one of the allowed lowercase letters to *Char1*.

Situation 2

The key pressed is a non-alphabetical character ([, for example), but is not an "action" key (such as RETURN, BACKSPACE, or an arrow key). This is a *SpecKeyCA* in the list of key types (4a.). In this case (with one exception), the mapping is based on the scan code of the key. Though the key may be relabeled, the CONTROL-*Char* combination is *always* mapped based on the scan code of the key, using the following table. (All numbers are decimal):

Scan code	U.S. Kbd legend	Mapped value	New codename
3	2 @	0	Null
7	6 ^	30	RS
12*	- _	31	US
26	[{	27	Esc
27	] }	29	GS
43	\ |	28	FS

Note

The mapping for the hyphen character (-) is the one exception. The scan code for it is ignored, and only the ASCII code for hyphen (decimal 45) is looked for (in *Char1*) when mapping the CONTROL- "-" key combination. This is because there may be more than one occurrence of the "-" key on the keyboard.

Situation 3

The key pressed is an "action" key, such as RETURN, BACKSPACE, or an arrow key. These keys generate special values when used in conjunction with the CONTROL key. Those actions are defined in other notes where they apply.

Situation 4

The key pressed is a function key (F1–F12). Defined elsewhere.

Situation 5

The key pressed is an accent key. See "Note 9: Accent Key" for details.

Situation 6

The key is not defined in conjunction with CONTROL. In which case, the key is treated as Undefined, as described in "Note 4: Undefined Character."

Note 6: The Pad Key

The pad keys have several uses, depending on various shift states. Some of them are based on their position on the keyboard. Because keyboard layouts change, the following table lists the hard coded, assumed positions of the keypad keys, with the offset value that must be coded into *Char1* of the table. Any remapping must use the *Char1* values defined for the keys that correspond to the pad keys given by the U.S. keyboard legend or

Char2 values shown:

U.S. Kbd legend	Scan code	*Char1* required (Binary)	*Char2* U.S. Kbd (ASCII)
HOME 7	71	0	7
UP 8	72	1	8
PGUP 9	73	2	9
—	74	3	
Left 4	75	4	
5	76	5	
Right 6	77	6	
+	78	7	
End 1	79	8	
5	76	5	
Right 6	77	6	
+	78	7	
End 1	79	8	
Down 2	80	9	
PGDN 3	81	10	3
INS 0	82	11	0
DEL .	83	12	.

Note

When NUMLOCK is off, or if SHIFT is active and NUMLOCK is on, the code returned is the "extended" code corresponding to the legends above (HOME, PGUP, etc). When NUMLOCK is on, or if SHIFT is active and NUMLOCK is off, the code returned is *Char2*. Note that the plus (+) and minus (—) keys will return *Char2* under *all* SHIFT combinations except with the ALT key.

When the ALT key is used with the *PadKeys*, the absolute value of the pressed key (determined by using the required *Char1* value) is added to the accumulated value of any previous numeric keys pressed with the ALT key. Before adding the new number to the accumulated value, that

accumulation is multiplied by ten, with any overflow beyond 255 ignored. When ALT is released, the accumulation becomes a character code, and is passed along with a scan code of zero. Note that if *any* key other than the 10 numeric keys is pressed, the accumulated value is reset to zero.

When *AltGraphics* is used with *PadKeys*, the *Char3* value is returned if it is nonzero and if an *AltGraf* bit is set in *XTableFlags1*; otherwise, it is treated the same as the ALT key.

Note 7: State Key

Each entry for a state key has as *Char1* a mask having the bit within the keyboard Shift Flags' lower byte to be set when the key is recognized. In the case of toggle keys, that bit will be toggled after each on the first make after pressing the key. The others will be set on the make and cleared on the break. *Char2* must have the bit to set in the upper byte of the Shift Flags when the key is down. *Char3* is used in place of *Char2* when the secondary key prefix is recognized immediately prior to this key. The masks are as follows (numbers are in hexadecimal):

Key	*Char1*	*Char2*	*Char3*
Right SHIFT	01	00	00
Left SHIFT	02	00	00
CONTROL-SHIFT	04	01	04
ALT-SHIFT	08	02	08
SCROLL LOCK	10	10	10
NUMLOCK	20	20	20
CAPSLOCK	40	40	40
SYSREQ	80	80	

Note that the INSERT key is *not* treated as a state key, but as a pad key. Also note that SYSREQ is included here since it is treated as a SHIFT key.

Note 8: ALT Keys

Most of the keys defined in a category that allows the ALT key (*AlphaKey*, *SpecKeyA*, *SpecKeyCA*) return a value called an *extended character*. This value is a character code of 00H or E0H with a second byte (using the *ScanCode* field of the *CharData* record) defining the extended code. In most cases, this value is the scan code of the key. Since the legend on these keys may be remapped on a foreign language keyboard, the ALT-based

extended code is hard to define in a general sense. So the following three rules are used:

1. *AlphaKey*: The extended code is derived from *Char1* (the lowercase character) as it was originally mapped on the IBM PC keyboard. The original scan code value itself is the extended code that a character will return. These keys can be moved and will still return their original ALT extended codes.
2. *SpecKeyA* and *SpecKeyCA*: This category is used for all keys that are not alphabetical characters or action codes (such as the RETURN or BACKSPACE key. The only exception is the TAB key, which *is* treated as a character).

   On foreign keyboards, these keys may be moved around and/or have new values assigned to them (such as special punctuation symbols), so the ALT mappings *must* be based on the *real* scan code. The effect of this is that keys defined by the *SpecKeyxx* classification will only have an ALT mapping if it is in one of the positions defined in the following table, in which case, the ALT extended code is as shown in the table:

Scan code	U.S. Kbd legend	ALT value
2	1 !	120
3	2 @	121
4	3 #	122
5	4 $	123
6	5 %	124
7	6 ^	125
8	7 &	126
9	8 *	127
10	9 (	128
11	0)	129
12	- _	130
13	= +	131

3. *FuncKey*: See "Note 3: Key Types" for the definition.

   When *AltGraphics* is used, the *Char3* value is returned if it is nonzero and if an *AltGraf* bit is set in *XTableFlags1*; otherwise, it is treated the same as the ALT key.

Note 9: Accent Keys

When an accent key is pressed along with CONTROL or ALT, it is treated as a regular key. The character it is translated to is the one in the *CtlAccent* or *AltAccent* field of the *AccentEntry* pointed to by the *Char1* value of the *KeyDef*. If the key being defined will have no defined value with CONTROL or ALT, it should have zeros in the field of the undefined combination.

When an accent key is pressed by itself (or with right or left SHIFT or *AltGraphics*) it will not be translated immediately. The *Char1* (or *Char2*, when left or right SHIFT is used or when the *AltGraphics* is used) index in the *KeyDef* record will be used with the next key received to check if the next key has an accent mapping. If that next key has no mapping for this accent (that is, if it has no bit set in its *AccentFlags*, or if that next key is not found in this accent's *AccentEntry*), then the character value in the *NonAccent* field of the *AccentEntry* is used as the character to display, followed by the translation of that next key (that is, both characters are passed on) after the "not-an-accent" beep sounds.

Note that if a key does not change when a left or right SHIFT key is held down, it should use the same value for *Char1* and *Char2* so that the accent will apply in both the shifted and non-shifted cases. If the accent value is undefined when used with a SHIFT key or with *AltGraphics*, the value in *Char2* or *Char3* should be zero.

Any accent key that does not have an ALT or CONTROL mapping should put zeros in the *AltAccent* and *CtlAccent* fields of its *AccentEntry*.

Note 10: Extended State Keys

For special National Language Support (NLS), the *KeyDef* record of the keyboard device driver maintains an additional byte of shift status. Key types 12H and 13H manipulate that byte. The other fields of the *KeyDef* are as follows:

Field	Description
Char1	A mask in which the bits that are on define the field used for the *Char2* value. Only the bits in the *National Language Support shift status* byte that correspond to the bits in this byte will be altered by the *Char2* value.

Char2 For KeyType 12H (Extended Shift), this is the value to OR into the byte when the make code is recognized and whose inverted value is ANDed when the break code is recognized. For KeyType 13H (Extended Toggle), this is the value XORed into the byte on each make code recognized (break code ignored).

Char3 Used in place of the *Char2* when the secondary key prefix (hex E0) is recognized immediately prior to pressing this key.

Examples of usage are as follows: *Char1* and *Char2* can define single shift status bits to set/clear/toggle. *Char2* can be a set of coded bits (delineated by *Char1*) that will be set to a numeric value when the key is pressed and cleared to zero when released (or on the next press if toggle). When *Char1* has all bits on, the whole byte can be set to *Char2*.

Note 11: The SPACEBAR

The key treated as the space character (SPACEBAR) should have a flag set in its *AccentFlags* field for each possible accent (that is, for each defined *AccentEntry* in the *AccentTable*). Each *AccentEntry* should have the space character defined as one of its accented characters, with the translation being to the same value as the accent character itself. The reason for this is that, by definition, an accent key followed by the space character, maps to the accent character alone. If the table is not set up as described, a "not-an-accent" beep sounds whenever the accent key, followed by a space, is pressed.

Note

The SPACEBAR is defined as a *SpecKeyA* (type 4) because its use in conjunction with the ALT key is allowed. If used with ALT, it will still return the ASCII space character. It will also return the ASCII space character when used with the CONTROL key.

Note 12: Translate Table Size

The translate table described in this appendix has the following dimensions:

Xlate Header	= 40
127 *KeyDefs* @ 6 bytes	= 762
16-byte *CtlAltTble*	= 16
7 *AccentEntries* @ 46 bytes	= 322
Total	= 1140 bytes