Operating System/2™ Technical Reference

Volume 1

Programming Family

First Edition (September 1987)

Preface

This book contains technical information for the IBM Operating System/2™ (OS/2™[1]). It provides a comprehensive and detailed description of OS/2 interfaces. This book also provides descriptions of OS/2 architecture, control structures, data structures, and I/O formats necessary to understand and use those interfaces.

This book is intended for the experienced user, system programmer, and application developer. You should be knowledgeable about operating systems and be proficient in one or more of the IBM Personal Computer programming languages, as well as being familiar with the 80286 architecture.

Related Publications

IBM Operating System/2™ User's Reference
IBM Operating System/2™ Programmer's Guide
IBM Personal Computer *AT®[2] Technical Reference*
IBM Personal Computer *XT™[3] Model 286 Technical Reference*
IBM Personal System/2™ Model 50/IBM Personal System/2™ Model 60 Technical Reference
IBM Personal System/2™ Model 80 Technical Reference
IBM Personal Computer *BIOS Technical Reference*
IBM Personal Computer *Macro Assembler/2*
iAPX 286 Programmer's Reference Manual including the *iAPX 286 Numeric Supplement* 210498, or the *iAPX 386 Programmer's Reference Manual* (IBSN 1-55512-022-9) Literature Department, Intel®[4] Corporation, 3065 Bowers Avenue, Santa Clara, CA. 95051

---

1 IBM Operating System/2™ and OS/2 are trademarks of International Business Machines Corporation

2 Personal Computer AT is a registered trademark of International Business Machines Corporation

3 Personal Computer XT Model 286 and IBM Personal System/2™ are trademarks of International Business Machines Corporation

4 Intel® is a registered trademark of Intel Corporation.

Contents

Chapter 1. Introduction

OS/2 has three methods of providing system services to application, or subsystem, programs:

- API Function Requests
- IOCtl Functions
- DevHlp Services

The Application Programming Interface (API) function calls provide a method for an application, or subsystem, to request OS/2 services (function requests). API function requests are invoked by a Call-Return interface with the stack used to pass request parameters.

OS/2 device drivers are used by OS/2 to access the I/O hardware. The IOCtl functions provide a method for an application, or subsystem, to send device-specific control commands to a device driver. The IOCtl functions are issued through the DosDevIOCtl API function request. The IOCtl functions are sub-functions of the DosDevIOCtl API function request. (For a detailed description of the IOCtl functions refer to *Technical Reference, Vol. 2.*)

The DevHlp services provide a method for OS/2 device drivers to request OS/2 services. Many OS/2 device driver functions are related to system operations rather than to hardware operations. The DevHlp functions are used to request OS/2 system functions, when writing your own OS/2 device driver.

OS/2 uses OS/2 device drivers to access I/O devices attached to an IBM Personal Computer AT® or Personal Computer XT™ Model 286 or an IBM Personal System/2™. An Application Programmer can write programs using the OS/2 device drivers supplied with OS/2, or his own OS/2 device driver.

API Function Requests

All OS/2 API function requests are invoked with a CALL interface. There are significant advantages when using a CALL programming interface if the parameters are "pushed" onto the stack before issuing the CALL. The hardware actually copies the parameters from the requestor's stack to the receiving program's stack, thus giving optimum addressability and protection at minimal execution cost.

OS/2 has a Dynamic Link facility by which linkage to a system function or routine is not resolved until run time. The Dynamic Link facility provides improved storage utilization because common library routines need not be linked to each load module. Programs written to the OS/2 interface are usually smaller in size (both on disk and in memory).

In addition, a restricted subset of the OS/2 API function requests is supported as a DOS *Family API*. Applications written to this API and linked with OS/2 library routines are capable of executing in either OS/2 or DOS modes.

The API Function Requests can be grouped into the following categories by function:

- Memory Management
- Process Control
- Session Management
- I/O Services
- Country Support

Memory Management

OS/2 memory management allows an application to use the 80286 extended memory (physical storage up to 16Mb). OS/2 memory management allows a user to concurrently execute more applications than fit in memory. Also, any single application and its data can be larger than real memory. OS/2 maintains the most active set of segments in memory at any one time by swapping the least active segments to disk, then reloading them when needed.

The memory management functions allow an application to:

- Allocate multiple data segments
- Implement an application as a number of distinct CALLable segments with OS/2 providing loading on demand as necessary
- Explicitly load applications if desired
- Package an application so that the linkage to library routines or infrequently used routines is not made until run time.

Process Control

OS/2 process control functions can be grouped into the following categories by function:

- Timer Services
- Multitasking
- Interprocess Communication.

Timer Services

In addition to the Date and Time functions provided in DOS, OS/2 provides the following functions:

- Regularly occurring intervals
- Asynchronous intervals
- Sleep for a period of time.

Multitasking

The multitasking functions of OS/2 allow a user to operate several applications concurrently. For most purposes, each application appears to have the entire computer to itself and may be designed and coded in much the same manner as with DOS. An application can also be designed such that its functions are divided among a collection of cooperating processes.

Multitasking is an integral part of OS/2. A priority-based, time-sharing scheduler is provided with special consideration being offered for applications with time-critical response requirements. Functions are provided to:

- Start new processes
- Start/stop/modify execution threads within a process
- Coordinate execution among several processes.

Interprocess Communication

The interprocess communications (IPC) functions allow processes to communicate with one another. These functions allow a program to:

- Communicate between processes via pipes, signals, queues, semaphores, and shared memory
- Explicitly control other processes execution
- Control access to serially reusable resources
- Signal occurrence of a flag event.

Session Management

The session management functions of OS/2 allow an application to select, set status, start, and stop sessions.

I/O Services

OS/2 provides the following I/O services to applications:

- I/O function calls
- Code Page support
- Device Monitor Services
- Printer/Spooler Services

OS/2 provides access to I/O through function calls and device drivers. Some devices are accessed through function calls specific to the device, such as the keyboard (KBD), mouse (MOU), and video I/O (VIO) calls.

Code Page Support

OS/2 Code Page Support allows a user to select a code page for keyboard input and screen and printer output before running an application, a system command or utility in the OS/2 multitasking environment. This allows the user in a particular country such as England or Norway or a language region such as Canadian-French to run with a code page that defines an ASCII-based character set containing characters used by that particular country or language.

Device Monitor Services

Character Device Monitors provide a mechanism for applications or subsystems to monitor all characters passing through a device driver. This mechanism allows any registered process to remove, insert or modify the information passing through the device.

Keystroke Monitor: A keystroke monitor can pass the keystroke through, consume the keystroke, or replace the keystroke with one or many keystrokes. Some applications monitor all keystrokes and provide global system function before more conventional applications receive the keystrokes. Examples include national language support for switching the keyboard layout and for Asian language input conversion.

Printer/Spooler Services

OS/2 provides printer/spooler services. The primary purpose of the spooler is to accumulate data directed to a printer on a per session basis. When the application is complete, the data will be output to the printer in one contiguous data stream. This reduces the possibility of intermixing printed output from different sessions to the same printer. The spooler also has the capability of invoking the Code Page Switcher which provides functions to allow an application to control code pages and fonts.

DevHlp Services

Device driver helper routines are provided for managing the request queue, blocking and unblocking, locking and unlocking memory.

Access to these system services is obtained at the time of device driver initialization. The request packet for the INIT command contains a pointer to the DevHlp interface. The pointer to the DevHlp interface is a bimodal pointer; that is, this pointer to the DevHlp interface is valid in both real mode and protect mode. The device driver does not have to be sensitive to the mode of operation before requesting DevHlp services.

A DevHlp 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 driver initialization time.

OS/2 Device Driver Architecture

OS/2 device drivers are divided into two parts - a strategy routine and an interrupt routine:

- The strategy routine is called with an I/O packet which describes the request. The strategy routine marks the request incomplete and queues the request. If the device is not busy it starts the device. Then it returns to the kernel which typically blocks on the incomplete I/O packet.
- The interrupt routine services the I/O completion. If there is new work in the queue, it starts the device. Then it indicates that the previous operation is complete and unblocks any threads which are waiting for this request to be completed.

OS/2 Device Drivers

The device drivers provided with OS/2 service requests in both the DOS mode and the OS/2 mode. Where appropriate, OS/2 device drivers provide a queued request interface rather than the serial request design of DOS device drivers. OS/2 device drivers support multitasking.

The device drivers supplied with OS/2 are:

- Asynchronous Communication (RS-232)
- Mouse Device Drivers
- Pointer Draw Device Driver
- VDisk
- CLOCK$
- Console (Screen and Keyboard)
- Screen (OS/2 mode)
- Keyboard (OS/2 mode)
- EGA Register Interface (DOS mode)
- Diskette
- Fixed Disk
- Printer

Problem Determination

OS/2 provides a system trace for problem determination.

Country Support Considerations

Country Support for OS/2 includes these features:

- Country Dependent Information
- Country APIs
- National Keyboard Layouts
- Configuration Commands
- Translation of System Message Files

Chapter 2. Application Program Interface (API)

The OS/2 Application Programming Interface (API) represents requests for system services. The API functions are invoked by a CALL-RETURN interface with the stack being used to pass the request parameters. The most obvious benefits are:

- Less need for a high-level language system services library — the respective function call may be interfaced directly from a high-level language such as IBM C/2™[1].
- Optimum performance — the target routine may be invoked directly rather than having an intermediary "router" get control first.
- The same interface mechanism is available for invoking an OS/2 routine as well as a library routine.
- Function replacement is an architected and well-defined activity.

When using the multiple protection rings of the 80286, there are significant advantages to a CALL programming interface if the parameters are "pushed" onto the stack before issuing the CALL. The hardware actually copies the parameters from the requestor's stack to the receiving program's stack, thus giving addressability and protection at minimal execution cost. All OS/2 functions are invoked via the CALL interface. A means to similarly call system extensions and I/O privilege routines executing at protection ring 2 is also provided. The application interface to the OS/2 mode and device drivers is strictly hierarchical, and the 80286 hardware supports and enforces this hierarchy.

---

[1] C/2 is a trademark of International Business Machines Corporation.

Dynamic Linking

The 80286 protect mode CALL architecture also offers benefits of greater importance than hierarchical structure and data copying. In particular, the following are definite advantages over the typical static module structure of DOS:

- Application programs need only load the most commonly used segments when started. Exception processing routines do not need to be loaded. They can be called (and be automatically loaded by the system) as necessary.
- Dynamic link package updates can be transparent to their clients. Existing applications can use enhanced function calls in a dynamic link package, and the existing applications need not change.

The actual programming steps required to use the dynamic link feature are the same as those of a static environment. The steps are:

1. The programmer codes a call to a subroutine which is to be dynamically linked and declares it "EXTERNAL FAR".
2. The compiler generates a standard external reference.
3. When the object module is linked, the linker is provided with the names of libraries containing dynamic link definition records. These records provide correspondence between the called entry point and the module file containing the routine being called.

PC Family API

In developing a single product to be used on either OS/2 or DOS 3.3, application developers can use a subset of the full function OS/2 API. This *subset* is the DOS Family API. An application that is written to this subset functions on either system.

The interface to the target operating system is provided as a set of program modules. These modules are loaded only when executing in a DOS mode.

Note: After compiling or assembling and linking for OS/2, you must BIND your .EXE file in order to make it run in both the OS/2 and DOS modes.

OS/2 Function Calls

This section describes how OS/2 function calls are issued. OS/2 applications must use the dynamic link mechanism (FAR CALL) to get to all services. The old style (INT 21H ...) function calls are supported only for the DOS mode.

OS/2 Function Call Rules

Rules for the OS/2 interface are shown below:

Rule 1: All parameters are passed on the stack (SS:SP).

Remarks Passing parameters on the stack is consistent across a broad base of languages on the 80286 family of processors. This method allows direct access to the operating system from high order languages. The minimum recommended stack space available is 2K bytes.

Rule 2: All interfaces pass a return code back to the caller in AX.

Remarks The use of register AX as a function return code is also consistent across many languages. All user registers except the FLAGS register are preserved. The contents of the FLAGS register are undefined. The state of the direction flag in the FLAGS register is preserved. If the direction flag was clear when an API was called then it will be clear when the API returns.

Rule 3: All addresses of OUTPUT parameters are of the form:

`selector:offset.`

Remarks Fully qualified addresses are available across all memory models as a method for returning values to a requestor. This allows one function entry point to serve all languages and memory models.

Rule 4: Each function is accessed by a FAR CALL.

Remarks This is a requirement for the functions to be dynamic link entries.

Rule 5: All functions remove the parameters from the stack.

Remarks Parameter lists are fixed length on a function basis. Variable length parameter lists are not supported.

Rule 6: All function names must be upper case at link time.

Remarks If a compiler or assembler generates case-sensitive (upper and lower case) external references, all calls or function definitions must be in all UPPER case characters.

OS/2 Function Call Characteristics

The function call descriptions follow a pseudo assembly language format. The interfaces are shown to be descriptive rather than to be an example of a coding sequence. The conventions described below are employed throughout this book.

Function Call Format

Because all parameters are pushed onto the stack, there are several pseudo instructions to describe these operations.

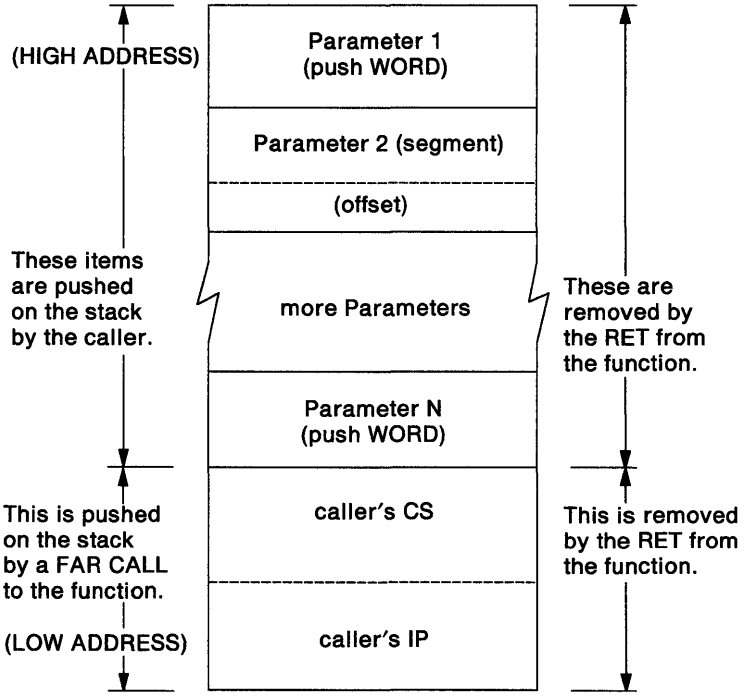
- PUSH - push an item onto the stack
  This can be used to push various size items onto the stack. The data item types are described below.
- PUSH@ - push the address of an item onto the stack
  All addresses in these interfaces are composed of a 32-bit value: a 16-bit selector, and a 16-bit offset. This address can point to any of the data item types.
- CALL - call a function
  All function calls are accessed via FAR CALLs. This is a requirement to utilize the Dynamic Link mechanism.

The following are data item types used to make up a parameter list:

- WORD - 2 bytes
  This type of operand can be passed by value (pushed onto the stack) or by reference (the address of the operand is passed on the stack).
- DWORD - 4 bytes
  This type of operand can be passed by value (pushed onto the stack) or by reference (the address of the operand is passed on the stack).
- ASCIIZ - null (0) terminated character string
  This type of operand can be accessed only by reference.
- Other - any other structure
  This type of operand can be accessed only by reference.

Interface Stack Frame

All parameters are passed via the stack. An example stack frame is shown below:



The following is the calling sequence that corresponds to the diagram above.

```
PUSH    WORD    Parameter 1
PUSH@           Parameter 2
•
•
•
PUSH    WORD    Parameter N
CALL            function
```

Function Calling Sequence Example

This is a sample function call. In *Technical Reference, Vol. 2*, the function calling sequence is shown using pseudo code. The pseudo code is similar to IBM Personal Computer Macro Assembler. Some definitions and necessary statements for segments and procedures are left out. The following example shows how a function call builds a stack and also how the pseudo code relates to actual code.

Pseudo Code	Actual Code

`extrn DOSXAMPL:far`	`extrn DOSXAMPL:far`
	
	`;`
	`; these items are in DS`
	`;`
	`IN1  dw  44`
	`IN2  db  'x'`
	
	`OUT1  dw  0`
	`.`
	`.`
	`.`

`PUSH WORD IN1`	`push IN1`

`PUSH@ WORD OUT1`	`push ds`
	`mov  ax,offset OUT1`
	`push ax`

`PUSH WORD IN2`	`mov al,IN2`
	`xor  ah,ah`
	`push ax`

`CALL DOSXAMPL`	`call DOSXAMPL`

OS/2 Sample Functions

The following are sample functions. The code indicated is similar to IBM Personal Computer Macro Assembler. Some definitions and necessary statements for segments and procedures are left out.

Function DOSXAMPL

This is an example function that retrieves information from a stack. It shows entry and exit sequences for a function and how to access the parameters.

```
push bp
mov bp,sp
.
.
.
les bx,[bp+8]           ; get the @ of OUT1

mov word ptr es:[bx],77     ; put 77 in OUT1

mov ax,[bp+12]          ; put IN1 in ax

mov bl,[bp+6]           ; put IN2 in bl
.
.
.
mov ax,17               ; set the return code to 17

pop bp
ret (far) 8             ; remove parameters from stack
                        ; and return
```

High-Level Language Interface Examples

These are examples of high-level language interfaces for the "OS/2 Sample Functions" on page 2-8 and the "Function Calling Sequence Example" on page 2-7.

IBM Pascal Compiler/2™[2]: This is an example of the interface used with the IBM Pascal Compiler/2™.

```
Declaration        function DOSXAMPL(    p_in1  :integer;
                                       var p_out1 :integer;
                                           p_in2  :char
                                     ): integer;

Data Declaration   var p_in1,
                       p_out1 :integer;
                       rc     :integer;
                       p_in2  :char;

Invocation         rc := DOSXAMPL(44,out1,xin2);
```

IBM C/2™[3]: This is an example of the interface used with the IBM C/2™.

```
Declaration        int far pascal DOSXAMPL();

Data Declaration   int_out1;

                   int rc;

                   char xin2;

Invocation         rc = DOSXAMPL(44,(char far *) &out1,xin2);
```

---

[2] Pascal Compiler/2 is a trademark of International Business Machines Corporation.

[3] C/2 is a trademark of International Business Machines Corporation.

OS/2 Compatibility Considerations

Three levels of Application Programming Interface (API) are supported by OS/2 to provide different levels of function and compatibility:

- OS/2 dynamic link full-function API for programs that require multitasking and memory management function.
- DOS Family API (subset of OS/2 API) for programs that require compatibility with both OS/2 and DOS.
- DOS interrupt-based API for programs that require only DOS compatibility.

OS/2 Application Environments

Two application environments are supported by OS/2 for applications:

- The OS/2 mode
- The DOS mode

OS/2 provides two environments for running applications; the DOS mode and the OS/2 mode. The DOS mode allows DOS applications to run concurrently with OS/2 applications. The OS/2 mode allows multiple OS/2 applications (both full-function and family) to run concurrently.

The 80286 processor provides two hardware modes of operation, real mode and protected mode. To support the two application environments, OS/2 handles requests for service from both modes. OS/2 applications run in the protected mode of the processor.

A DOS application running in the DOS mode runs in both modes, that is, the processor switches from real to protected mode and back again. For example, all file requests are handled in protected mode. This provides a means of insuring data integrity across multiple applications in a multitasking operating system.

The following table shows the API support in each environment:

	Environment		
API	**OS/2**	**DOS**	**DOS 3.3**
OS/2	Yes	No	No
Family	Yes	Yes[1]	Yes[1]
DOS	No	Yes[2]	Yes[2]

Notes:

1. See the API Functions in *Technical Reference, Vol. 2* for Family API restrictions.
2. For DOS API compatibility exceptions in the DOS mode see "DOS Mode Exceptions" on page 2-19.

DOS Family and Full Function API

The DOS Family API function calls are a "subset" of the full OS/2 API function calls. When an application program is written using only the DOS Family API function calls, it can be linked to work with both OS/2 and DOS 3.3. A family application can also work under TopView.

In reading the chart that follows, remember:

- "Family API" (middle column) is a list of the family API functions that work in the OS/2 mode, the DOS mode, and DOS 3.3.
- The "OS/2 Only" column means these function calls work in the OS/2 mode only.
- The combination of both columns is the OS/2 full-function API.
- The function calls listed in the "Family API" column marked with an asterisk (*) are supported, but with restrictions. The restrictions are described in the detailed description for each affected function call in *Technical Reference, Vol. 2* as Family API Considerations.

Function Type	Family API Subset	OS/2 Only
Tasking		
		DosCreateThread
		DosCWait
		DosEnterCritSec
	* DosExecPgm	
	* DosExit	
		DosExitCritSec
		DosExitList
		DosGetInfoSeg
		DosGetPrty
		DosKillProcess
		DosPtrace
		DosSetPrty
Asynchronous Notification		
	* DosHoldSignal	
	* DosSetSigHandler	
Interprocess Communication		
		DosCloseQueue
		DosCloseSem
		DosCreateQueue
		DosCreateSem
		DosFlagProcess
		DosMakePipe
		DosMuxSemWait
		DosOpenQueue
		DosOpenSem
		DosPeekQueue
		DosPurgeQueue
		DosQueryQueue
		DosReadQueue
		DosResumeThread
		DosSemClear
		DosSemRequest
		DosSemSet
		DosSemSetWait

Function Type	Family API Subset	OS/2 Only
		DosSemWait
		DosSuspendThread
		DosWriteQueue
Timer		
	DosGetDateTime	
	DosSetDateTime	
	* DosSleep	
		DosTimerAsync
		DosTimerStart
		DosTimerStop
Memory Management		
	* DosAllocSeg	
		DosAllocShrSeg
	* DosAllocHuge	
	* DosCreateCSAlias	
	DosFreeSeg	
	DosGetHugeShift	
		DosGetShrSeg
		DosGetSeg
		DosGiveSeg
		DosLockSeg
		DosMemAvail
	* DosReallocHuge	
	* DosReallocSeg	
	DosSubAlloc	
	DosSubFree	
	DosSubSet	
		DosUnlockSeg
Dynamic Linking		
		DosFreeModule
		DosGetModHandle
		DosGetModName
		DosGetProcAddr
		DosLoadModule

Function Type	Family API Subset	OS/2 Only
Family API		
	DosGetMachineMode	
	BadDynLink	
Device Monitors		
		DosMonClose
		DosMonOpen
		DosMonRead
		DosMonReg
		DosMonWrite
Session Management		
		DosStartSession
		DosStopSession
		DosSelectSession
		DosSetSession
Device I/O Services		
	DosBeep	
		DosCLIAccess
	DosDevConfig	
	* DosDevIOCtl	
	DosGetPID	
		DosPFSActivate
		DosPFSCloseUser
		DosPFSInit
		DosPFSQueryAct
		DosPFSVerifyFont
		DosPhysicalDisk
		DosPortAccess
		DosSendSignal
		KbdDeRegister
	KbdCharIn	
		KbdClose
	KbdFlushBuffer	
		KbdFreeFocus
		KbdGetCp
		KbdGetFocus
	KbdGetStatus	

Function Type	Family API Subset	OS/2 Only
		KbdOpen
	* KbdPeek	
		KbdRegister
		KbdSetCp
		KbdSetCustXt
		KbdSetFgnd
	KbdSetStatus	
	KbdStringIn	
		KbdSynch
		KbdXlate
		MouClose
		MouDeRegister
		MouDrawPtr
		MouFlushQue
		MouGetDevStatus
		MouGetEventMask
		MouGetNumButtons
		MouGetNumMickeys
		MouGetNumQueEl
		MouGetPtrPos
		MouGetPtrShape
		MouGetScaleFact
		MouInitReal
		MouOpen
		MouReadEventQue
		MouRegister
		MouRemovePtr
		MouSetDevStatus
		MouSetEventMask
		MouSetPtrPos
		MouSetPtrShape
		MouSetScaleFact
		MouSynch
		VioDeRegister
		VioEndPopUp
		VioGetAnsi
		VioGetBuf

Function Type	Family API Subset	OS/2 Only
		VioGetCp
	VioGetConfig	
	VioGetCurPos	
	VioGetCurType	
	VioGetFont	
	VioGetMode	
	VioGetPhysBuf	
	VioGetState	
		VioModeUndo
		VioModeWait
		VioPopUp
		VioPrtSc
		VioPrtScToggle
	VioReadCellStr	
	VioReadCharStr	
		VioRegister
		VioSavRedrawUndo
		VioSavRedrawWait
	* VioScrLock	
	VioScrollDn	
	VioScrollUp	
	VioScrollLf	
	VioScrollRt	
	* VioScrUnLock	
		VioSetAnsi
		VioSetCp
	VioSetCurPos	
	VioSetCurType	
	VioSetFont	
	VioSetMode	
	VioSetState	
		VioShowBuf
	VioWrtCellStr	
	VioWrtCharStr	
	VioWrtCharStrAtt	
	VioWrtNAttr	
	VioWrtNCell	

Function Type	Family API Subset	OS/2 Only
	VioWrtNChar	
	VioWrtTTY	
File I/O		
	DosBufReset	
	DosChDir	
	DosChgFilePtr	
	DosClose	
	DosDelete	
	DosDupHandle	
	* DosFileLocks	
	* DosFindClose	
	* DosFindFirst	
	* DosFindNext	
	DosMkDir	
	DosMove	
	DosNewSize	
	* DosOpen	
	DosQCurDir	
	DosQCurDisk	
	* DosQFHandState	
	DosQFileInfo	
	DosQFileMode	
	DosQFsInfo	
	DosQHandType	
	DosQVerify	
	DosRead	
		DosReadAsync
	DosRmDir	
		DosScanEnv
		DosSearchPath
	DosSelectDisk	
	* DosSetFHandState	
	DosSetFileInfo	
	DosSetFileMode	
	DosSetFsInfo	
		DosSetMaxFH

Function Type	Family API Subset	OS/2 Only
	DosSetVerify	
	DosWrite	
		DosWriteAsync
Errors and Exceptions		
	DosErrClass	
	* DosError	
	* DosSetVec	
Messages		
	* DosGetMessage	
	DosInsMessage	
	DosPutMessage	
Trace		
Program Startup		
	DosGetEnv	
	DosGetVersion	
Code Page Support		
	* DosGetCp	
	DosSetCp	
		DosSetProcCp
Country Support		
	DosCaseMap	
	DosGetCollate	
	* DosGetCtryInfo	
	DosGetDBCSEv	

Note: BadDynLink is a call generated internally by the BIND utility. BadDynLink is for dynamic link calls that are unresolved in DOS 3.3 and yet valid in the OS/2 mode. This allows applications to have a set of function calls available in the OS/2 mode that are not available in DOS 3.3. The DosGetMachineMode call can be used in the application to determine at run-time whether a call is appropriate.

If a routine is called in DOS 3.3 that is not valid for the DOS mode, the BadDynLink routine aborts the application.

DOS Mode Exceptions

In the DOS mode, all the functions that are supported in DOS 3.3 with SHARE installed are supported in OS/2 with some exceptions.

Note: Undocumented DOS 3.3 function calls are not supported.

DOS 3.3 Network Function Calls

No DOS 3.3 Network functions calls are supported.

Version Number

The version number returned to an application in response to the version call is 10.0, instead of the displayed version number 1.0.

Background Freezing

When the process (the DOS mode application) is in a background session, the DOS mode is frozen. The process receives no task-time CPU service and no interrupts. One effect is that an application that is tracking the time of day by counting clock tics will have an incorrect count when it returns to the foreground.

Hooking Hardware Interrupts

DOS mode applications may hook any hardware interrupt vector unless the interrupt is already owned by an OS/2 device driver in which case the DOS mode application is terminated. If the keyboard interrupt is hooked by an application, it is shared between the OS/2 device driver and the DOS mode.

Direct Device Manipulation

There are restrictions on the devices DOS mode applications may directly manipulate:

- 8042 Keyboard Interrupt Controller - DOS mode applications may not reprogram the 8042 keyboard interrupt controller.
- 8253 Clock/Timer chip - DOS mode applications may reprogram this at will.
- 8259 interrupt controller - DOS mode applications may not reprogram the 8259 interrupt controller.
- Disk Controller - DOS mode applications may not reprogram the disk controller(s), although they may have direct access via INT13, INT25 and INT26. However, INT26 is not allowed to non-removable media and for INT 13H, only

functions 01 (read status), 02 (read sectors), 0A (read long), and 15 (read DASD type) are allowed to non-removable media.

- DMA Controller(s) - DOS mode applications may not reprogram the DMA ports.
- COM/Parallel Port - DOS mode applications should not use COM ports that are being used by OS/2 mode applications, or by the Print Spooler utility.

  Using a port in the DOS mode removes it from availability in the OS/2 mode. If the OS/2 mode uses one of these ports, it is unavailable to the DOS mode. Applications that perform I/O directly to the serial and parallel ports without checking the BIOS presence indicators can interfere with the operation of these devices by the OS/2 bimodal device drivers.

  DOS mode applications will fail if they attempt to access the BIOS 40: data area when the Asynchronous device driver is loaded if SETCOM40 has not been run. AUX is no longer supported as an alias for COM1.

Device Drivers

See "Compatibility with Previous-Level Device Drivers" on page 7-29.

DOS Software Interrupts

See "DOS Execution Environment Software Interrupt Support" on page 7-31 for exceptions.

DOS Commands

See "DOS Commands Compatibility Exceptions" in the *OS/2 User's Reference*.

Chapter 3. Memory Management

Memory management under OS/2 using the Protected Virtual-Address mode of the 80286 permits applications to allocate more memory than could fit in the 640K physical limit that existed in previous versions of DOS. In addition, if the user enables swapping, OS/2 allows one or more applications to allocate more memory than is physically available in the system.

Use of the Segmentation Hardware

A feature of OS/2 is utilization of memory beyond the 640K limit through comprehensive memory management functions. These functions fully exploit the segmentation capability of the hardware in the protected address mode of the 80286. Use of these functions allows full utilization of the processor architectural storage limit of 16Mb.

Applications written with these memory management functions will be able to use more memory than the 640Kb limit of previous DOS versions. When memory becomes full, OS/2 maintains the most active segments in storage. Inactive segments (least recently used segments) are swapped to disk. Active segments are loaded into storage as needed.

These new OS/2 mode functions permit you to concurrently execute more programs than will fit in memory. Any single program and its data can be larger than real memory.

The new functions provided allow a program to:

- Allocate a large number of data segments; each may be up to 64Kb.
- Allocate multiple memory segments of the maximum segment size (64Kb).
- Keep each segment private or share it with other programs.
- Package an application so that the linkage to library routines or infrequently used routines is not made until run time.
- Implement an application as a number of distinct CALLable segments with OS/2 providing load on demand.
- Explicitly load programs if desired.

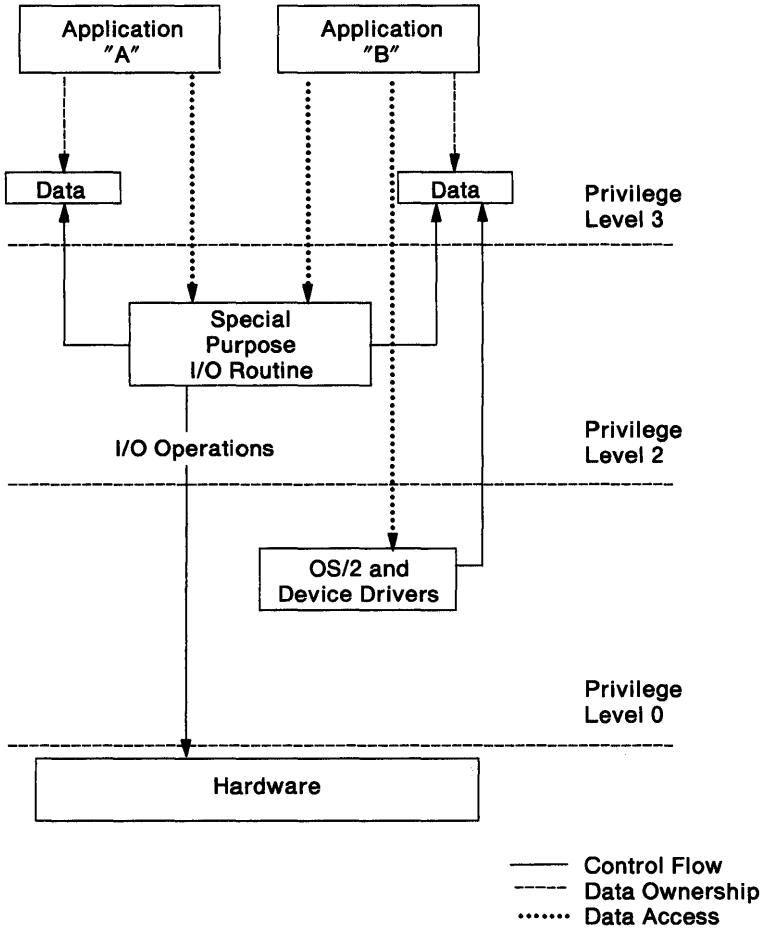
The Protection Features of the Ring Structure

The OS/2 protection model takes advantage of the 80286 ring protection architecture to protect the integrity of applications.

The protection levels used by OS/2 for program execution are:

- Applications run at protection level 3.
- Special purpose routines (other than general device drivers) requiring I/O privilege execute at protection level 2.
- The kernel and device drivers run at protection level 0.

The following diagram shows the application perspective of the protection model provided by OS/2:



For a module to be loaded at protection level 2, the CONFIG.SYS IOPL=yes command must indicate this is to be allowed. Protection level 2 modules cannot use dynamic link, and cannot make OS/2 function calls.

The OS/2 protection model uses the 80286 ring protection architecture to insure that for each level in the hierarchy, only programs at that level, or a more privileged level, can access data at that level. For example:

- An application has access only to its own data at level 3.
- A level 2 I/O module has access to its clients' application data at level 3.
- The kernel and device drivers have access to data at all levels.

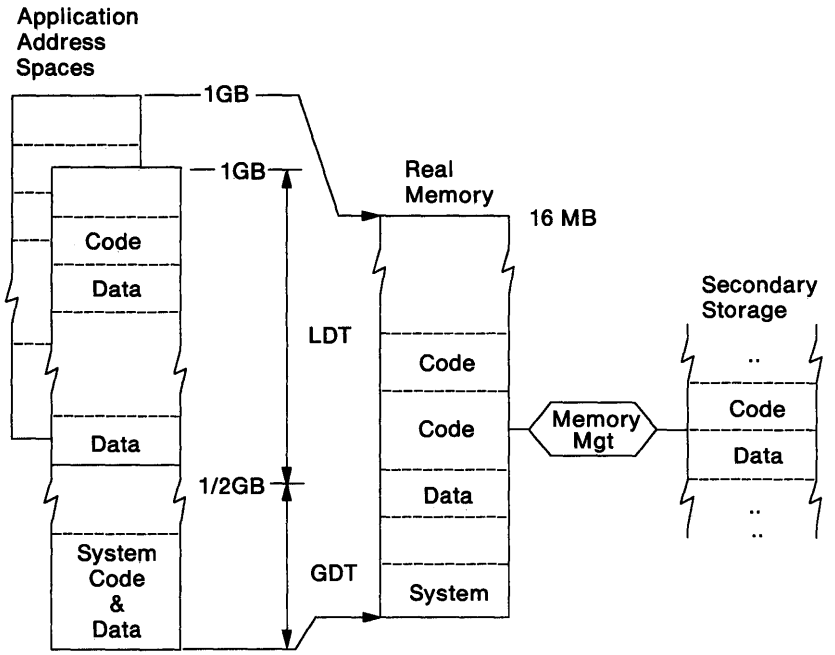
OS/2 Mode Memory Management

Protect mode memory management enables an OS/2 mode application to use large real memory beyond the 640Kb boundary imposed by previous versions of DOS. Memory management is supported with both memory compaction and segment swapping.

In protect mode there is one Local Descriptor Table (LDT) per process for all threads running under that process. Each process has its own distinct address space, mapped by an LDT and the Global InfoSeg. The LDT provides a per process private address space while the Global InfoSeg provides addressability for system-wide data and programs which are shared among all processes.

Together these tables provide a virtual address space of 1Gb. However, the available virtual address space is limited by the available disk storage space. The LDT and Global InfoSeg provide a mapping from this 1Gb virtual address space to the 16Mb (maximum) physical address space of the 80286. OS/2 dynamic link routines have LDT, as opposed to Global InfoSeg, descriptors. Reference "Dynamic Linking" on page 4-26 for more information on dynamic link routines.

The following is a diagram of the segment swapping, protected memory management of OS/2.

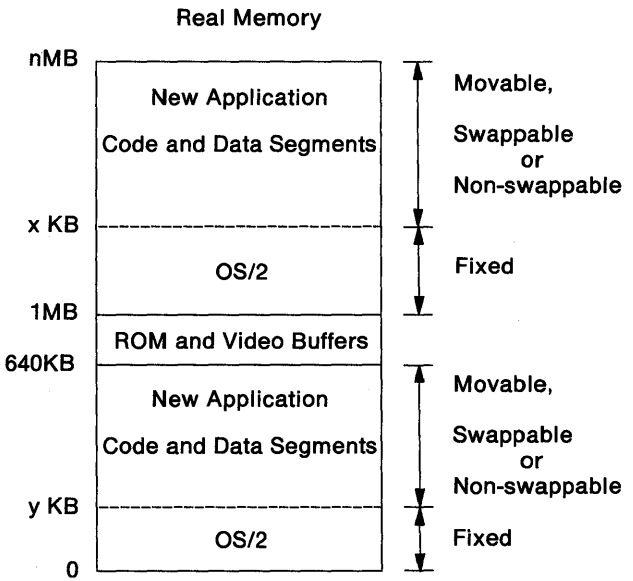


The features of protect mode memory management support include:

- Storage over-commitment - the amount of storage allocated at any instant for data and code segments can be greater than the amount of real memory available.
- Segment discard - real memory occupied by code segments that are still in the address space of an application (though not currently in use) can be reclaimed by discarding the segment. When the discarded segment is later referenced, a fresh copy is read from the .EXE file, whereas swapping restores from a swap area.
- Segment motion (memory compaction) - because segments are variable length, real memory is subject to external fragmentation. Holes of deallocated real memory, each of which is insufficient in size to satisfy a request for memory, but which together would be sufficient, are united in order to satisfy the request.

- Protection - applications only have addressability to memory segments specifically authorized to them by the system. The OS/2 system and each individual application are protected from access by other applications.

Real Memory Map (Protect mode only)



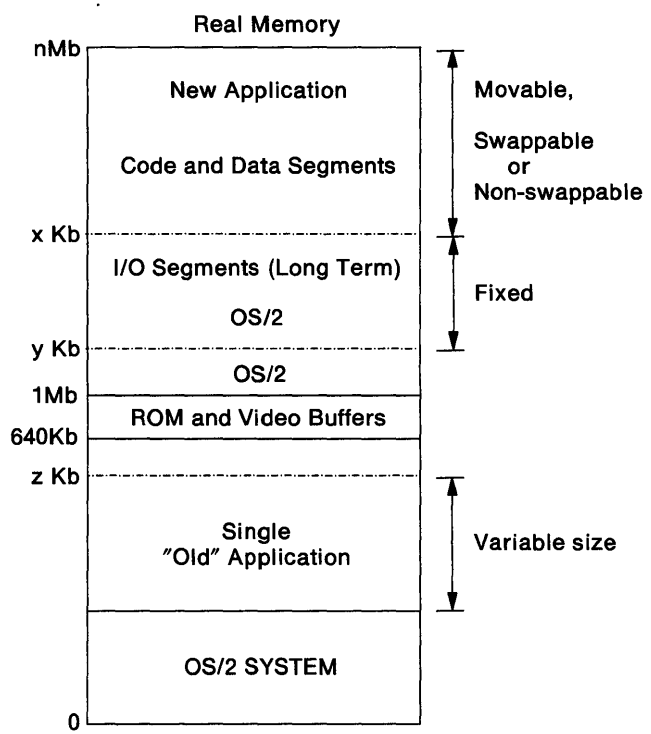
Note the x Kb and y Kb boundaries. These are movable boundaries separating the fixed and movable regions of memory. The actual location of these boundaries will vary depending on the system load and the amount of real memory installed.

Real Mode Memory Map

A real mode system allows execution of a single DOS mode application in addition to new OS/2 mode applications.

A DOS mode application executes only below 1Mb (in fact, because of the reserved ROM and video buffer space, only below 640Kb) while an OS/2 mode application may execute at any address.

The following diagram shows the memory layout for an OS/2 system running both OS/2 mode applications and a single DOS mode application. OS/2 mode code and data segments are loaded above a boundary set by the RMSIZE= command in CONFIG.SYS. The following diagram assumes the boundary is set at 640K bytes.



Note the x Kb and y Kb boundaries. As in the previous Real Memory Map, these are movable boundaries separating the fixed and movable regions of memory. The actual location of these boundaries will vary over time, depending on the system load and the amount of real RAM installed.

Note the z Kb boundary. This defines the logical end of memory for the DOS 3.3 application and may vary up to the 640Kb limit. Protect mode memory is above this boundary.

Memory Management Function Call Summary

The following memory management interfaces are supported: (For a detailed description of these function calls refer to *Technical Reference, Vol. 2.*)

DosAllocHuge	Allocates multiple memory segments of the maximum segment size (64Kb).
DosAllocSeg	Allocates a segment of memory to the requesting process.
DosAllocShrSeg	Allocates a shared memory segment.
DosCreateCSAlias	Creates a code segment alias descriptor for a data segment passed as input.
DosFreeSeg	Deallocates a segment.
DosGetHugeShift	Returns a shift count. The shift count is used in deriving the selectors to address memory allocated with DosAllocHuge.
DosGetSeg	Gets access to a shared memory segment.
DosGetShrSeg	Allows a process to access a shared memory segment previously allocated by another process.
DosGiveSeg	Gives a memory segment to another process.
DosLockSeg	Locks a discardable segment in memory.
DosMemAvail	Returns the size of the largest block of free memory.
DosReallocHuge	Changes the size of memory allocated by DosAllocHuge.
DosReallocSeg	Changes the size of a segment already allocated.
DosUnlockSeg	Unlocks a discardable segment.

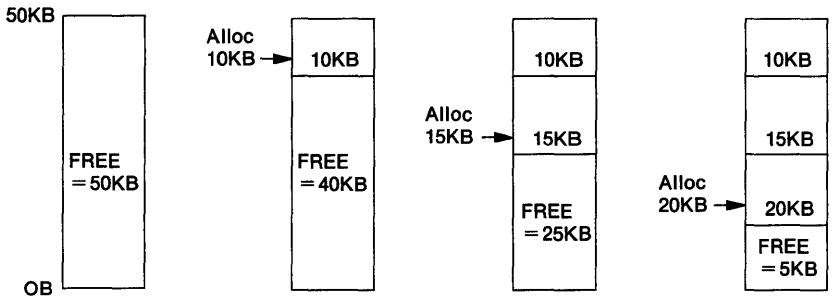
Memory Suballocation Package (MSP)

In addition to the extensive segment swapping memory management functions described above, OS/2 includes a high-performance mechanism for suballocation within a segment (small model) similar to a linked-list of storage descriptors.

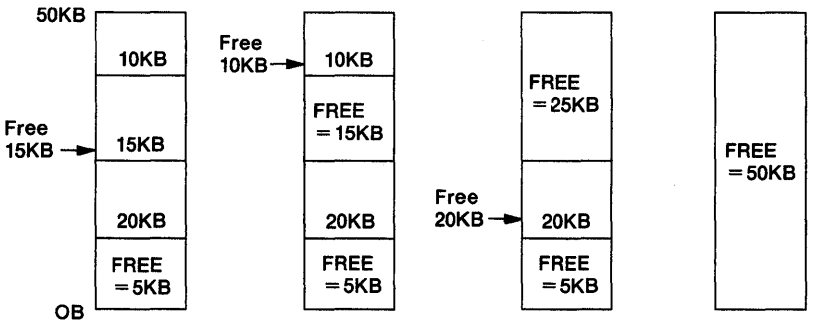
The OS/2 Memory Suballocation Package (MSP) is a set of intrasegment memory allocation dynamic link routines. The functions support memory suballocation within a segment. The memory suballocation dynamic link routines are packaged within the dynamic link module DOSCALL1.DLL. External references to memory suballocation dynamic link routines are resolved at link time.

Memory Suballocation Example

Following is an example of an application making memory suballocation requests. At the top is a succession of allocate requests which deplete the FREE storage within the segment.



The application may now begin freeing the storage as it is no longer needed. Adjacent free blocks are combined:



MSP Function Call Summary

The following MSP interfaces are supported:
(For a detailed description of these function calls refer to *Technical Reference, Vol. 2.*)

DosSubSet Used either to initialize a segment for sub-allocation or to change the size of a suballocation segment previously initialized.

DosSubAlloc Allocates memory from a segment previously allocated by DosAllocSeg or Dos AllocShrSeg and initialized by DosSubSet.

DosSubFree Frees memory previously allocated by DosSubAlloc.

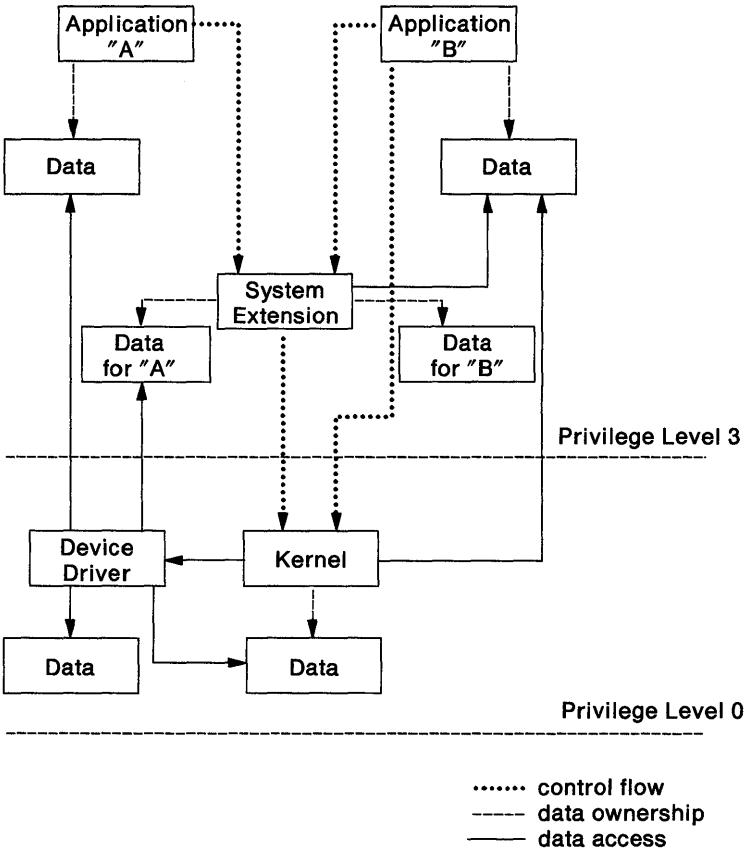
System Extensions

OS/2 provides a base on which application developers can construct solutions to more complex applications. To an application, these solutions typically appear as an extension of the operating system (for example, Presentation Manager); and they require many of the same capabilities as the kernel, such as protection, data isolation and sharing, and an efficient means of invocation. The design of OS/2 provides functions that allow the extension author to obtain these capabilities easily.

The system extension provides a callable routine which is its application interface. The actual connection from the application program to the system extension is made at run time (see "Dynamic Linking" on page 2-2 and "Dynamic Linking" on page 4-26 for details on this linkage.) When called, the extension routine either performs the request directly or passes the request on to a separate process which performs the request asynchronously. System extensions are implemented as dynamic link modules.

The determination of processing directly or under a separate process must be made based on the data isolation and performance characteristics of the solution being offered.

The following diagram shows the invocation of a system extension via the run time dynamic linking.



For processing of the request under a separate process, interprocess communications techniques may be implemented using semaphores, queues, signals, or shared memory.

The above diagram also demonstrates the program execution flow and the data handling aspects of providing a system extension as a callable routine. Note that the extension may allocate data exclusive to each of its clients.

Chapter 4. Process Control

The process control functions of OS/2 are:

- Timer Services
- Multitasking
- Interprocess Communication (IPC)
- Asynchronous Notification
- Program Execution Control
- Errors and Exceptions
- OS/2 Message Functions

Timer Services

These services are related to time of day and intervals of time. They provide the full range of time primitives for implementing timer-based applications.

Timer Management

In OS/2, all time-related functions are based on a periodically interrupting time source. The timer operates with a frequency of approximately 32 hertz. This rate is sufficient for most applications, but you are limited to a time interval with a precision of less than 50 milliseconds.

Time/Date

DosGetDateTime and DosSetDateTime are the function calls that get and set the system date and time. (For a detailed description of these function calls refer to *Technical Reference, Vol. 2.*)

Timer Intervals

In addition to the Date and Time functions, OS/2 provides the following time-interval functions:

- Asynchronous intervals - the system notifies a task that a period of time has elapsed.
- Regularly occurring intervals - the system continuously notifies a task that a designated period of time has elapsed.
- Sleep for a period of time - a task can delay its execution for a certain period of time.

The timer interval functions allow specification of the time interval in milliseconds; however to reduce system overhead, the actual resolution is typically on the order of 50 milliseconds.

The timing functions are only accurate to within one or two clock ticks. The application can perceive that the time elapsed is longer than specified because of delays in getting the application resumed. All time values are in milliseconds and are rounded up to the next clock tick. The clock tick duration can be determined by using the DosGetInfoSeg function and examining the timer interval field in the Global InfoSeg.

If it is necessary for an application to know the time elapsed while waiting for a timer, a comparison can be made of the milliseconds field (in the Global InfoSeg) before the wait began and the milliseconds field after the application resumed execution. This technique is accurate to within one or two clock ticks. The value in the milliseconds field will roll over every few weeks. If the application suspended prior to a rollover and awoke after a rollover then the elapsed time calculation may need to take that into account.

When interrupts are disabled for periods longer than the clock tick interval, the milliseconds field can lose time. However, the time of day (hours, minutes, seconds), time in seconds since 1-1-70, and the date will remain accurate. When interrupts must be disabled for a long period, the elapsed time should be computed from seconds, rather than milliseconds.

Timer Services Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The timer services function calls provided are summarized as follows:

DosGetDateTime	Get the system date and time
DosSetDateTime	Set the system date and time
DosSleep	Synchronous pause (wait or sleep) for interval of time
DosTimerAsync	Start asynchronous timer, one shot
DosTimerStart	Start interval timer, asynchronous, continuous
DosTimerStop	Stop interval or asynchronous timer

Multitasking

Multitasking is an integral part of OS/2. A priority-based, time-slicing scheduler is provided with special consideration for applications with time critical response requirements.

The functions provided to a multitasking application developer include the ability to:

- Initiate and terminate other processes
- Vary a process' dispatch priority
- Execute programs as separate processes
- Coordinate execution among several processes
- Communicate between several processes.

Many of these functions are also available for multiple execution threads within a single process.

The multitasking features of OS/2 allow a user to operate several applications concurrently. For most purposes, each application appears to run in a separate *protected* computer by itself and can be designed and coded in much the same manner as under the current DOS. This case of multiple concurrent applications is the simplest form of multitasking. In this instance, each application's execution is managed under a *process* and one *thread* of execution.

For more complex application requirements, an application can be designed such that its functions are divided among a collection of cooperating processes (or threads).

Threads are dispatched for execution on a priority basis with a time-slicing scheduler being provided to ensure threads of equal priority receive an equal opportunity to execute. For applications with time critical response requirements, special mechanisms (such as DosEnterCritSec and DosExitCritSec) are provided to ensure their successful operation.

Tasking (Processes and Threads)

This section describes those functions that are provided to initiate and terminate processes, to vary process priority, and to execute other programs as separate processes.

The section "Interprocess Communication (IPC)" on page 4-9 describes those functions that provide coordination and communication between processes.

Definitions

Process An instance of program execution. Includes at least one thread and ownership of resources defined or used by the program.

Thread A unit of execution within a process. A thread is the dispatched unit.

A process is that identifiable entity which represents an executing program and the resources in use by that program. Only a process can own resources; a thread can only access resources on behalf of its parent process.

The thread is the dispatched unit within a process. A thread is not identifiable outside its parent process.

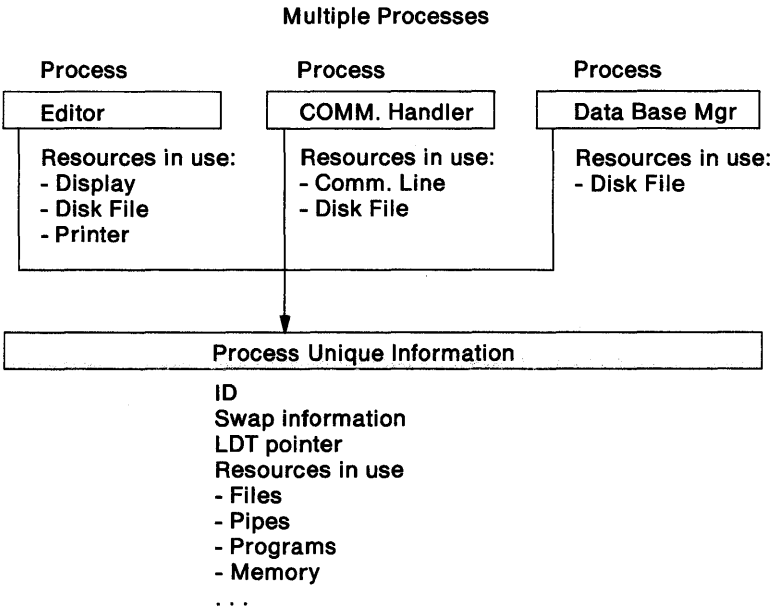
Where necessary, various sections of this book use terms such as current thread, waiting thread, or parent thread. This terminology is meant to clarify discussing one thread of a process, rather than the process itself.

An application can be designed as several distinct processes or as multiple threads within a single process. Consider the following:

- The creation and termination of a thread is very fast while the creation of a process is relatively slow.
- Sharing of data and resources between threads is natural. Sharing between processes requires special considerations.
- The creation of a process is costly to system memory.
- If the problem can be best solved by multitasking entities with a high degree of independence, multiple processes is the proper choice.
- Several independent processes may also be the best choice for applications in which the independence or function of the process model is required. This is particularly true when the various programs of the application are tightly coupled with respect to interprocess communication or fixed, shared data areas.
- Multiple threads is the better choice if the concurrent execution entities are started for only a short period of time as the overhead to start and end a thread is much less than for a process.
- A thread is also more suitable for problems in which multiple executions are needed, yet where each execution need not be externally identifiable and does not require distinct or separate resources.

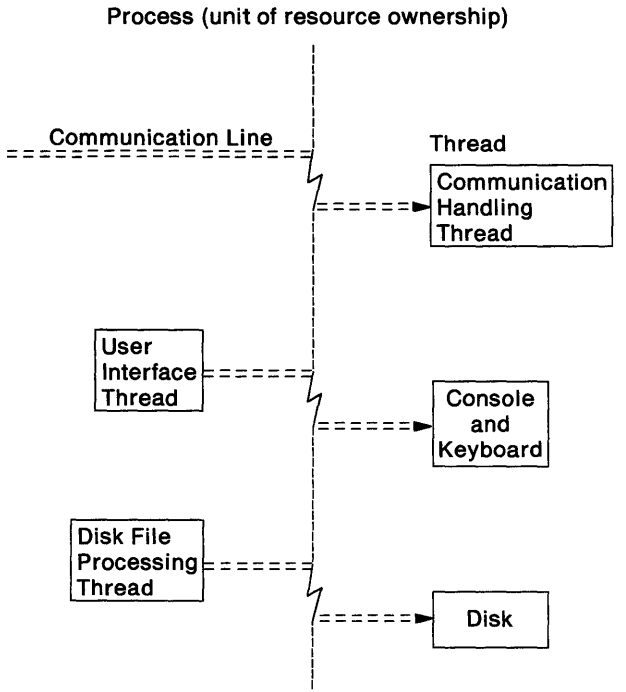
The diagram below shows the process structure when the user has started three applications: an editor, a communications handler, and a data base manager. Because these are independent applications, they have no knowledge of one another. The fact that they can share a physical disk on which their individual *Disk Files* are located is known and managed only by OS/2.

Multiple Independent Processes Diagram



For a solution to an application where multiple asynchronous execution threads are needed, the following diagram demonstrates how the various threads might operate, each using different devices to accomplish a portion of the overall solution.

Multiple Threads within a Process Diagram



Process-Unique Information
- ID
- Swap information
- LDT pointer
- Resources in use
  - Files
  - Pipes
  - Programs
  - Memory
- .
- .
- .

Thread-Unique Information
- ID
- Disparching Information
- Priority
- Processor state
- Time slice
- Stack pointer
- Registers

Resource Management

OS/2 provides resource management and tracks ownership at the process level. The resources which a process can own, or be granted controlled access to, are:

- Files (and devices if opened at that level)
- Memory
- Pipes
- Queues
- System semaphores

When a process terminates, any of the above resources which it does not explicitly close or release is released automatically.

Tasking Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The tasking function calls are summarized as follows:

DosCreateThread	Start a thread of execution within a process.
DosCwait	Wait for a child process's termination.
DosEnterCritSec	Enter critical section of execution.
DosExecPgm	Execute a program as a new process.
DosExit	Exit the current thread or process.
DosExitCritSec	Exit critical section of execution.
DosGetInfoSeg	Get addresses of system variables segments.
DosGetPrty	Get a process's (or thread's) priority.
DosKillProcess	Terminate another process.
DosSetPrty	Set a process's (or thread's) priority.

Interprocess Communication (IPC)

OS/2 provides several methods for interprocess communication:

- Signals
- Messages
  - Pipes
  - Queues
- Semaphores
  - RAM semaphores
  - System semaphores
- Shared memory

Pipes, queues, and semaphores are created by the applications. They are identified by handles, or addresses that are passed among the interested processes as necessary.

The semaphore support provides serialization, or signalling, by means of RAM semaphores and system semaphores:

- RAM semaphores are a high-performance mechanism best used between the threads *within* a process.
- System semaphores are a high-function mechanism particularly suited for use *between* processes.

To ease the task of sharing resources between programs, OS/2 extends the standardized naming conventions of the file system to queues, system semaphores, and shared memory. This ensures references to the same name are resolved to the same resource.

Communication via Signals

Communication via signals is used to allow one process to set an external event flag to another process. The target process must use DosSetSigHandler to inform OS/2 that it wishes to intercept any of three flags. Another process then can issue a DosFlagProcess indicating which of the flags to signal. The target process will receive control at the signal handler it has defined for that signal.

Communication via Messages

Two facilities provide for interprocess communication via messages - pipes and queues.

Pipes

Pipes are an IPC mechanism based on the file I/O concept. Pipes are a technique by which two related processes can communicate as if they were doing file I/O. In fact, a program which inherits a pipe handle cannot distinguish if its I/O requests to that handle are to a file or pipe.

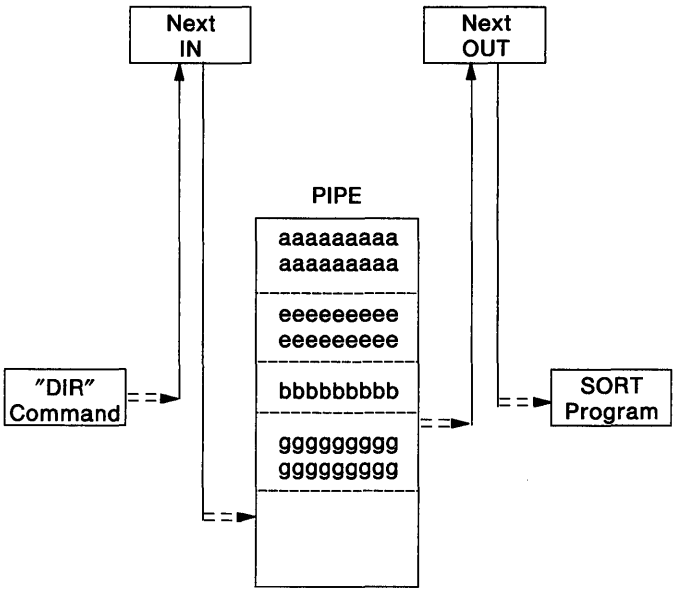
For communicating via pipes, the standard OS/2 read and write functions are used. Pipe support is provided only for applications in which the pipe participants are a closely related group of processes. The functions provided are in the DosMakePipe function call.

The storage required, or available, for a pipe I/O request to be performed can be a consideration. Pipes are effectively fixed length in nature. The most any pipe can hold is 64Kb at any one time. If a pipe is full, further write requests will block until sufficient data is removed from the pipe.

For example, with COMMAND.COM a user issues a DIR command, piping DIR's output to SORT. This gives a sorted directory listing.

Piping Output of One Program to Another: The following diagram depicts how information is piped from one program to another:

Command: DIR | SORT

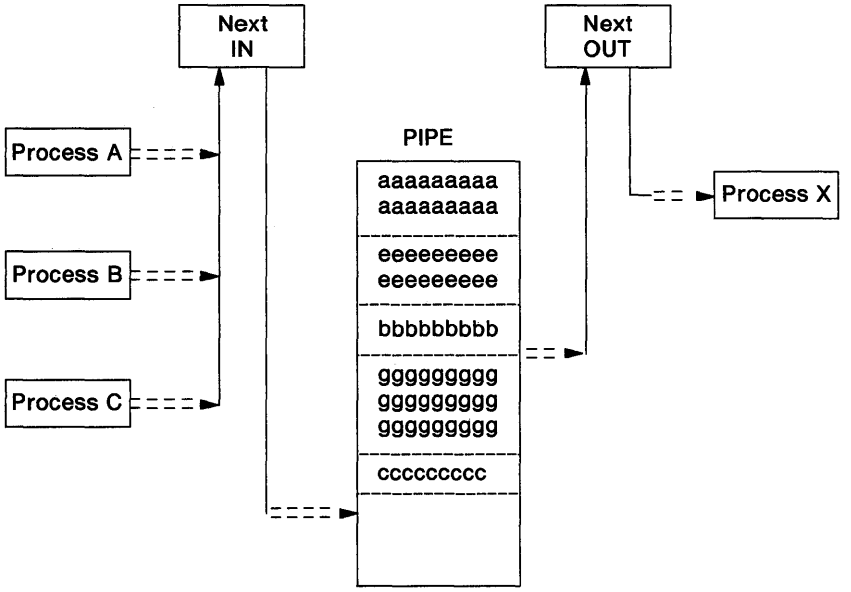


OS/2 keeps track of the data and free space in the pipe with the Next IN and Next OUT pointers. When the pipe is full, the next write by the DIR process waits until SORT has removed enough data to make room for the new message.

Pipes can also be used as a form of a fixed length first-in-first-out (FIFO) circular queue providing communication between processes.

Communicating with a Pipe

The following diagram depicts the use of a pipe to pass data to a server process X from three child processes A, B, and C. The sending processes send data independently of one another. Process X removes data from the pipe in the order in which it arrived.



Comparing Pipes with Files:

- Similarities
  - Data is communicated to pipes by the standard DosRead and DosWrite function calls.
  - Pipes are closed by the standard DosClose function call.
- Differences
  - Pipes are created by DosMakePipe rather than one of the file system create requests.
  - Pipes need not be opened before being accessed, only the DosMakePipe is necessary.
  - Pipes are implemented via an internal storage buffer mechanism rather than having their data maintained on disk.

- When writing to a file, the requesting thread is blocked only while doing file I/O. When writing to a pipe, the requesting thread will block if the pipe reader allows the pipe to fill up.
- Writing to a pipe is not interspersed. No other thread can write to that pipe until the write in progress is completed. Any thread that attempts to write to the pipe is blocked.
- Reading data from a pipe removes that data from the pipe. Subsequent reading will not find that data.

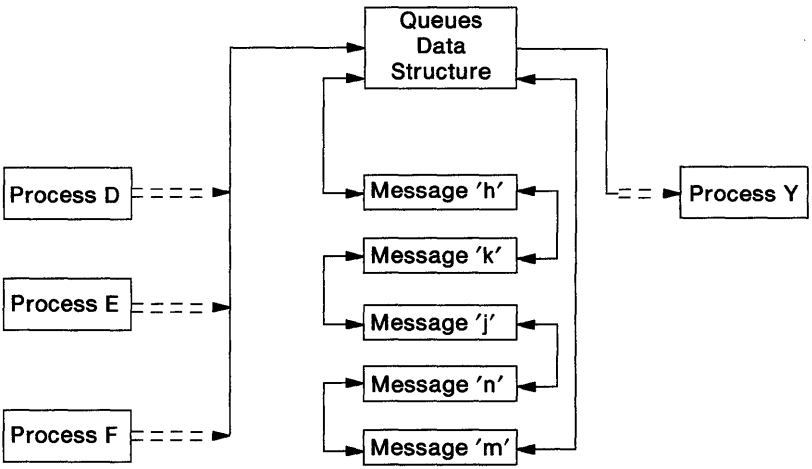
Pipes are inherited in the same manner as files. A using process typically creates a pipe, then starts a child process (which inherits the pipe handles) and communicates to the child process with the pipe handles.

Queues

For more sophisticated applications, queues provide a more powerful mechanism for interprocess communication of data between processes.

Communicating with a Queue

The following diagram shows a queue passing data to a server process Y from three child processes D, E, and F.



As with pipes, the sending processes can send data independently of one another. However, unique to queues, outstanding elements can be ordered by priority or by arrival order, first-in first-out (FIFO) or last-in first-out (LIFO). Also, process Y can examine each element in the queue and remove them whenever, and in any order, desired.

A feature of queues compared to pipes is that queues have a performance advantage because the data is not copied in queues, but is passed in a shared segment. Also, there is virtually no size limitation for the messages themselves. While the total message text a pipe can contain is 64Kb, queues can contain very large amounts of data. This is because each message is a unique block of storage and the aggregate of all messages can be dispersed across the entire machine.

Comparing Pipes and Queues

Queues are similar to pipes, with the following exceptions:

- Pipes use the file system interface (close, read, write). Queues have their own special calls.
- Pipes are data-stream oriented; queues contain message packets.
- Pipes are FIFO; queue messages can be ordered FIFO, LIFO, or by priority and can be accessed in random order.
- Data in queues can be purged.
- Queue data is not copied; the shared segment containing the data must be made accessible to the recipient.
- There can be multiple writers into a queue, but only one reader/owner. Multiple readers can be realized by means of multiple threads within the process which owns a queue.
- A count of the outstanding queue messages can be supplied to any process which has access to a queue.
- Queues are variable length; however, the maximum number of elements which can be placed in a single queue is no greater than 3260.

Like pipes, queues can be blocked on.

Pipes are a technique used by two processes to communicate as if they were doing file I/O. In fact, a program which lists a file can have its input or output specified as being a pipe and the program would operate without change. The pipe support would ensure that all data read or written by the program went to a pipe rather than to a file or a printer.

Programs using queues must be designed and coded with the concepts and function calls defined by queuing. These calls are not like the file I/O calls, and the processing of data is completely different from pipes.

Queues are more efficient than pipes because, only a pointer to the data is passed.

The storage required, or available, for the IPC to be performed is often a consideration. Pipes are fixed in length (not over 64Kb); while queues can be of relatively unbounded length because queue messages do not need to be contained in one segment.

The application designer must take into account all these factors balancing the performance requirements against the storage use characteristics of each solution.

Queue Function Calls

For function call details, refer to *Technical Reference, Vol. 2.*

The queue function calls are summarized as follows:

DosCloseQueue	Close a Queue
DosCreateQueue	Create a Queue
DosOpenQueue	Open a Queue
DosPeekQueue	Get an element from queue, but do not remove it
DosPurgeQueue	Purge all entries from a queue
DosQueryQueue	Find how many elements are in queue
DosReadQueue	Get an element from a queue and remove it
DosWriteQueue	Add an element to a queue

Managing Queues

This section is an aid to application programmers who are implementing more complex server environments with OS/2 queuing as the means of interface between processes.

In these environments, it may be advisable to provide a dynamic link interface routine which fields application program requests in a language consistent with the server's other calls and translates the requests into the appropriate OS/2 queuing and memory management calls.

For any of the techniques described here, the memory suballocation functions (DosSubAlloc, DosSubFree, and DosSubSet) should be used to allocate or free any message storage required.

The Segment for Data Storage

The OS/2 memory protection features provide total isolation between processes with memory being shared only when explicitly requested either by the shared memory function calls DosAllocShrSeg and DosGetShrSeg or by DosAllocSeg and DosGiveSeg or by DosGetSeg.

When using the OS/2 queuing functions, the two processes involved in the queuing conversation must each have addressability to the data that is communicated. (The queuing support provides the offset and length of the data, but not message copy facilities.)

Managing Data Storage Examples

Several techniques can be used to manage the data storage required for the message that is transmitted, depending on the requirements of the particular application. For example:

- For a solution which involves tightly coupled processes communicating with small messages, or at low data rates, manage the message storage in a single shared segment.

  The name for this segment can be established by mutual agreement among the parties involved or, for simplicity, the name chosen for this segment could be the same as that chosen for the queue.

The biggest disadvantage of this solution is the limit imposed by the single 64Kb segment size available for data elements; that is, 64Kb is the limit that can be devoted to message storage. This may not be a problem if the server process is never busy for a long period of time or if only short messages are being sent. But, the 64Kb limit may not be acceptable for long messages or when the server process is unable to keep up with requests from several clients. Remember, for this example, the 64Kb limit applies to the sum of all messages for all client processes.

- For a more complex server-client implementation, where potentially many client processes are driving a server, use a queue-driven server that provides support for several processes.

  For this example, assume the server provides either a single queue for all clients or a separate queue for each client.

  A table could be kept with each entry being composed of:

  - The Process ID (PID)
  - Shared memory segment name
  - Shared memory handle.

  On any CALL, the server need only obtain the currently active PID, scan its table looking for that PID and:

  - If found, get the storage handle to be used for this process.
  - If not found, the storage segment has not been created. As this must be the first request from this process, the shared segment must be created and a table entry setup for this process.

  When the server close function is received, the server must remove this process entry from the table.

- You could also allocate the data segment with DosAllocSeg and use DosGiveSeg to pass addressability to the server. This method allows each message to be any size up to the 64Kb limit.

  For this technique, the following steps can be used:

  - When issuing DosOpenQueue, the process ID of the queue owner is saved.
  - The queue element segment is allocated, via DosAllocSeg, specifying the segment is to be shared.

- DosGiveSeg is used to pass addressability to the segment from the client to the server.
- When a request is to be made, DosWriteQueue is issued referencing this segment by providing the Recipient SegHandle (returned from DosGiveSeg) in the DataBuffer field.

When the client completes its queue requests or terminates, the server should issue a DosFreeSeg to release the segment.

Queue Element Copy Function

A highly efficient element copy function can be implemented by means of the DosGiveSeg technique described above. As the client allocates the buffer and gives it to the server, there is no added performance cost required to subsequently make a copy of the data from the client's buffer to the server's buffer.

Queue Element Delete Function

Some server applications can require the ability to delete individual elements from a queue. For example, on discovering that a client process has terminated, it may be necessary to delete all remaining elements which are from that process.

This can be accomplished by setting up a loop similar to the following:

```
Do until end_of_queue
    DosPeekQueue NoWait
    If Have Element AND element's PID = PID of interest
      Then
          DosReadQueue Element Peeked above
Enddo
```

Coordinating Execution Among Several Threads

In a multitasking application environment, OS/2 functions allow processes to exercise some control over other processes and/or threads within their own process. The functions provided include:

- Semaphores (RAM-based and file-system-based)
- Starting and stopping a thread's execution

Semaphores

Semaphore functions allow multiple processes (or threads) to control access to serially reusable resources within the same process. There are two types of semaphores supported: RAM semaphores and system semaphores.

RAM Semaphores

RAM Semaphores are defined by the requesting program allocating a double word of storage and using the address in the semaphore calls provided below. RAM semaphores are a minimal-function mechanism with OS/2 performing no resource management services such as freeing when the owner terminates.

RAM semaphores are a high-performance mechanism that require using processes to have shared access to the same area of memory in which the RAM semaphore is defined. The affected processes define the semaphore by convention (mutual agreement) as a particular double word in some shared storage area.

RAM semaphores are best suited for signaling. They cannot be used to control access to resources reliably.

System Semaphores

System Semaphores are defined by OS/2 in response to a create-semaphore function call. Once created, they can be accessed by separate processes, and OS/2 provides full resource management, including freeing and notification when the owner terminates.

System semaphores are a full-function mechanism, providing control between any process and thread with OS/2 managing the storage for the semaphore data structure. System semaphores are defined within the file-system name space as a

pseudo file instead of a RAM location. The semaphore is a pseudo file in that its name takes the form of a file in the subdirectory SEM, although this subdirectory does not actually exist; system semaphores and their names are kept in memory.

Comparison of RAM and System Semaphores

When discussing semaphores, the terms owned, unowned, set, or clear are used to describe the state of a semaphore at a particular time. The concept of ownership applies only to system semaphores which are created with exclusive ownership indicated. When a semaphore is owned, it is an error for a thread other than the owner to try to modify that semaphore.

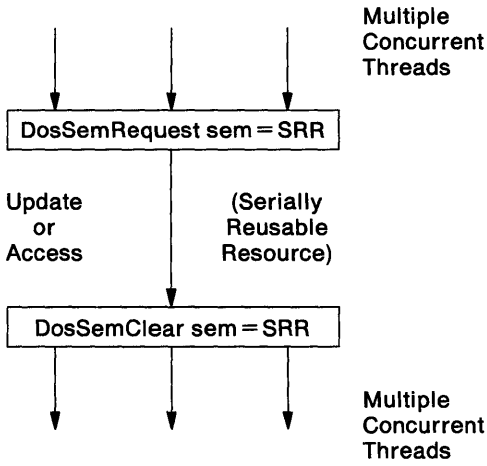
The handle for RAM semaphores is the address of the double word of storage allocated. If the address of the RAM semaphore is invalid, the system will terminate the process with a general protection fault.

- System semaphores are more flexible and easier to use than RAM semaphores because:
  - They can be used by processes which do not share memory.
  - Their ownership is relinquished when the owner terminates.
  - They provide more function.
- RAM semaphores offer a slight performance advantage over system semaphores because the handle used to refer to them is actually the address of the semaphore. The handle of a system semaphore must be translated to the memory address of the semaphore data structure.
- RAM semaphores should not be used between processes to control access to a serially reusable resource because it is not possible to relinquish the semaphore should the owner end abnormally. In the case of system semaphores, should the owner end, OS/2 will release the semaphore and notify the next thread which gets the semaphore that the owner ended while holding the semaphore.

  Note: RAM semaphores can be used between cooperating processes. DosExitList can be used, for instance, to write into a shared memory location to inform a process using a RAM semaphore that the semaphore owner has ended abnormally.

RAM Semaphore Diagram

The following diagram depicts the use of DosSemRequest and DosSemClear for serializing access to a resource. Only a single thread can enter the semaphore at a time. The effect is that all other threads are locked-out from use of the serially reusable resource until the entering thread leaves the semaphore.



General Semaphore Function Calls

The function calls provided for controlling access to a serially reusable resource (via either RAM or system semaphores) are:

DosSemClear Clear a semaphore.
DosSemRequest Obtain a semaphore.

Signalling via Semaphore Function Calls

In addition to the resource control semaphore function calls described previously, OS/2 provides three function calls to signal that an event has occurred between threads/processes: DosSemSet, DosSemSetWait, and DosMuxSemWait. These function calls can be used in combination with the DosSemClear and DosSemSet function calls to awaken a blocked thread whenever a semaphore is cleared, rather than when it is no longer owned. In fact, owning a semaphore, as mentioned in the DosSemRequest description, does not apply to these function calls.

To allow a simple wait or post type of signalling between threads/processes, several function calls are provided:

DosSemSet	Set a semaphore.
DosSemSetWait	Set a semaphore and wait for it to be cleared.
DosSemWait	Wait for a semaphore to be cleared.
DosMuxSemWait	Wait for any one of many semaphores to be cleared.

System Semaphore Function Calls

The function calls provided to support the allocation, access authorization, and deallocation of system semaphores are:

DosCloseSem	Close a system semaphore.
DosCreateSem	Create a system semaphore.
DosOpenSem	Open an existing system semaphore.

Starting and Stopping a Thread's Execution

For explicitly controlling when a thread can execute, the following function calls are provided:

DosResumeThread	Restart a thread's execution.
DosSuspendThread	Suspend a thread's execution.

IPC Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The Interprocess Communication function calls provided are summarized as follows:

DosCloseQueue	Close a Queue
DosCloseSem	Close a system semaphore.
DosCreateQueue	Create a Queue
DosCreateSem	Create a system semaphore.
DosMakePipe	Create a Pipe
DosMuxSemWait	Wait for any one of many semaphores to be cleared.
DosOpenQueue	Open a Queue
DosOpenSem	Open a system semaphore.
DosPeekQueue	Get an element from Queue, but do not remove it
DosPurgeQueue	Purge all entries from a Queue
DosQueryQueue	Find how many elements are in Queue
DosReadQueue	Get an element from a Queue and remove it
DosResumeThread	Restart a thread's execution.
DosSemClear	Clear a semaphore.
DosSemRequest	Obtain a semaphore.
DosSemSet	Set a semaphore.
DosSemSetWait	Set a semaphore and wait for it to be cleared.
DosSemWait	Wait for a semaphore to be cleared.
DosSuspendThread	Suspend a thread's execution.
DosWriteQueue	Add an element to a Queue

Asynchronous Notification

Signals allow a process to intercept and deal with a variety of traps and external events. The signal facility allows a program to specify an *on condition* handler routine which is executed when the event occurs.

Examples of events that can cause signal handlers to be executed are:

- Control-Break (or Control-C) key pressed
- Program terminated via DosKillProcess

Each process can define a process-unique signal handler for any signal.

Signals fall into two categories, traps and external events. Trap events are synchronous in that they are a result of an instruction executed by one of the process's threads. Most program traps (such as divide by 0) are handled via the DosSetVec function call. All other signals are external events.

An incoming signal is handled in one of several ways:

- The default action, typically IGNORE or TERMINATE PROCESS.
- If the process has specified a signal handling address and the exception is a trap, the thread causing the trap will execute the signal handler routine. The thread executes that handler immediately.
- If the process has specified a signal handling address and the exception is an external event, thread 1 (the original task thread) is diverted, in a forced far call analogous to a hardware interrupt, to the proper signal handler address. Because a signal represents a time-critical event, if thread 1 is in the midst of a function call that does not complete quickly then the function call is aborted. The signal interrupt takes place immediately upon return from the OS/2 service call. Slow calls aborted in this manner are primarily device I/O calls. File function calls (disk open/close/read/write) are not normally aborted.

An application which expects to make non-emergency use of signals should reserve thread 1, perhaps by having it block upon an eternally reserved RAM semaphore, and use another thread for program execution.

Asynchronous Notification Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The Asynchronous Notification function calls provided are summarized as follows:

DosHoldSignal	Disable/Enable signal processing
DosFlagProcess	Set process external event flag
DosSendSignal	Send CTL-C or CTL-Break signal
DosSetSigHandler	Define a routine to handle a signal.

Dynamic Linking

Dynamic linking is the delayed binding of external references. There are two forms of dynamic linking:

- Load time
- Run time.

Dynamic link routines, both load time and run time, are shared by invoking applications. Both the code and the data segments of a dynamic link routine are shared (unless the nonshared data option is selected during the module link process). It is a dynamic link routine's responsibility to serialize access to its shared data segments.

Note: A dynamic link routine can allocate nonshared memory.

In **load time dynamic linking**, a program calls a dynamically linked routine just as it would any external routine. When the program is assembled or compiled, a standard external reference is generated. At link time, the programmer specifies one or more libraries containing routines to satisfy external references. External routines to be dynamically linked contain special definition records in the library. A definition record tells the linker that the routine in question is to be dynamically linked and provides the linker with a dynamic link module name and entry name. The module name is the name of a special executable file with the filename extension of .DLL which contains dynamic link entry points. The linker stores module name or entry name pairs describing the dynamic link routines in the .EXE file created for the program. When the calling program is run, OS/2 loads the dynamic link routines from the modules specified and links the calling program to the called routines.

To invoke **run time dynamic linking**, the application uses the function calls described in this section.

A dynamic link module can have an optional initialization routine. This routine is invoked when the dynamic link module is first loaded, before any dynamic link routine is actually called. The initialization routine is the entry point specified on an END statement in one of the Assembler source files of the dynamic link module. Only one source

file can have an END statement identifying an initialization routine. If no entry point is specified on any END statement, no initialization routine is called. Input to the initialization routine is as follows:

AX = Module handle.

SI = HEAPSIZE parameter from the .EXE file.

DI = Module handle for the dynamic link module.

DS = The library's DGROUP data segment if one exists. Otherwise, DS = The application's DS.

Initialization routines exit via far return. Contrary to the standard convention, initialization routines set AX not equal to zero to indicate success and AX equal to zero to indicate failure.

Demand Load

The OS/2 demand load function supports loading code segments on demand when called during the execution of a program, as opposed to preloading all code segments before program execution begins. Code segments are discarded as required to provide space for other uses. Code segments which are demand loaded, as opposed to preloaded, are identified in the .EXE file header for the program.

Demand Load is enabled only when swapping is turned on.

Demand load is supported for the code segments of any program, not just for dynamic link routines.

I/O Privilege Model

Although OS/2 applications do not have I/O privilege, subsystems can have routines which execute with I/O privilege. Code (and data segments) requiring I/O privilege can be flagged when the subsystem is linked. Refer to "Dynamic Linking" on page 4-26.

A routine requiring I/O privilege is allocated a 512-byte stack. Segments executing with I/O privilege may not contain links to any other segments or modules. Such segments cannot issue OS/2 calls.

Starting programs which require I/O privilege can be controlled by the CONFIG.SYS parameter IOPL.

EXE File Information

The OS/2 .EXE file provides support for segmented programs and 80286 protect mode programs. An OS/2 .EXE file represents either an application or a module containing dynamic link entry points.

The following items are applicable to applications only:

- Segment #:offset of entry point
- Segment # passed in DS on program invocation
- Segment #:offset of stack
- Stack size

The following items are kept on a per segment basis:

- Code or data
- If code segment, pre-load versus demand load
- If data segment, whether segment is read only or read/write
- Whether segment requires I/O privilege
- Fix up records for external references including
  - References resolved within this .EXE file
  - References resolved via dynamic link. These include module name/entry point name pairs and/or module name/entry point ordinal pairs. Each pair identifies a dynamic link module (.DLL file) and a dynamic link routine within that module.
- Fix up records to support 80287 emulation
- Fix up records to support huge addressing
- Debug information.

The following items are kept per far call entry point:

- Entry point name
- Whether the entry point name is exported
- Segment #:offset of entry point
- The number of parameters which must be copied from the caller's stack to the callee's stack when an entry point requiring I/O privilege is invoked.

Program Execution Control Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The program execution control function calls provided are summarized as follows:

DosFreeModule	Free Dynamic Link Module
DosGetProcAddr	Get Dynamic Link Procedure Address
DosGetModHandle	Get Dynamic Link Module Handle
DosGetModName	Get Dynamic Link Module Name
DosLoadModule	Load Dynamic Link Module

Errors and Exceptions

Errors from Function Requests (Return Codes)

All function calls return AX = 0 if the operation is successful. If an error condition is encountered, AX = an error code.

For function call return codes and return code details, refer to *Technical Reference, Vol. 2.*

Hard Error Override

Hard error processing occurs without direct application notification. In the event that an OS/2 application needs to process these events, a new function call (DosError) is provided. This allows the application program to notify OS/2 that all permanent errors associated with the process, or an open handle belonging to the process, are to be reflected as immediate failures.

Handling Machine Exceptions

An application can provide a routine to process machine exceptions occurring while it is executing. This allows language libraries to furnish 287 emulation or error recovery routines.

Errors and Exceptions Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The errors and exceptions function calls provided are summarized as follows:

DosError	Allows a process to receive hard error notification without generating a hard error signal.
DosSetVec	Establish Handler for Exception Vector.

OS/2 Message Functions (Message Retriever)

The Message Retriever provides an application message functions to retrieve and insert variable message information, insert message text, and output messages.

Message Functions Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The function calls provided are summarized as follows:

DosGetMessage	Get a system message with variable text inserted.
DosInsMessage	Insert variable text string information into body of message.
DosPutMessage	Output a message to the supplied handle.

Program Startup Conventions

The rules on the content of various data areas and registers on program entry and termination are different for old programs than new programs.

For old programs, the conventions of DOS 3.3 apply.

For programs linked with OS/2, the new conventions are:

- There is no Program Segment Prefix (PSP)

- The environment, program pointer, and arguments are passed in a segment (selector in AX)

When the indicated program is given control, it will receive a pointer to a copy of the environment followed by a copy of the PgmPointer string followed by two argument strings. The following sample shows how to start a typical OS/2 program.

```
Environment:     ASCIIZ string 1    ; environment string 1
                 ASCIIZ string 2    ; environment string 2
                   ...
                 ASCIIZ string n    ; environment string n
                 Byte of 0
                   ...
Program Pointer: ASCIIZ             ; string of filename
                                    ; of program to run.
                   ...
Arguments:       ASCIIZ             ; argument string 1
                 ASCIIZ             ; argument string 2
                 Byte of 0
```

Environment consists of a list of strings typically having the following form:

```
parameter = value
```

Program Pointer is an ASCIIZ string of the drive, directory path, and filename of the program being executed.

Arguments consist of two strings representing command parameters for the program as opposed to the environment parameters. When a program is started from the command line, CMD.EXE will set argument string 1 to the program name as it was entered on the command line. Argument string 2 will contain program parameters as entered on the command line.

Shown below is a dump of a sample environment, program pointer, and argument area. The dump was obtained after entering the following command:

```
c:src\startup 1111 2222 3333 4444 5555 6666 7777 8888
```

Sample Dump

```
AX=004F  BX=003F  CX=00C3  DX=0000  SP=0200  BP=0000  SI=0000  DI=0088
IP=0000  CS=001F  DS=002F  ES=0000  SS=003F  NV UP EI PL NZ NA PO NC
GDTR=1172E0 3D07  IDTR=11B000 03FF   TR=0010  LDTR=0028 IOPL=2 MSW=PM EM TS
001F:0000 CC             INT   3

004F:0000  33 58 42 4F 58 3D 43 4F-4D 4D 41 4E 44 2E 43 4F   3XBOX=COMMAND.CO
004F:0010  4D 00 50 41 54 48 3D 00-43 4F 4D 53 50 45 43 3D   M.PATH=.COMSPEC=
004F:0020  41 3A 5C 43 4D 44 2E 45-58 45 00 00 43 3A 5C 53   A:\CMD.EXE..C:\S
004F:0030  52 43 5C 53 54 41 52 54-55 50 2E 45 58 45 00 63   RC\STARTUP.EXE.c
004F:0040  3A 73 72 63 5C 73 74 61-72 74 75 70 00 20 31 31   :src\startup. 11
004F:0050  31 31 20 32 32 32 32 20-33 33 33 33 20 34 34 34   11 2222 3333 444
004F:0060  34 20 35 35 35 35 20 36-36 36 36 20 37 37 37 37   4 5555 6666 7777
004F:0070  20 38 38 38 38 00 00 00-00 00 00 00 00 00 00 00   8888............
```

The following registers are set on program entry:

CS:IP Points to the program initial entry point specified in the .EXE header.

SS:SP Points to the stack specified in the .EXE header.

DS Points to the data segment specified in the .EXE header. When the program does not contain an automatic data segment, DS will contain zero and the program must initialize DS if necessary.

ES 0

AX Environment segment handle (selector).

BX Offset in environment of command line start

CX Length of data segment (0 = 65536)

DX STACKSIZE parameter from the .EXE file.

SI HEAPSIZE parameter from the .EXE file.

DI Module handle for the application executable.

BP 0

There are functions provided specifically for program startup. These functions allow an application to tailor its operation to specific versions of DOS. The function calls are:

DosGetEnv Get the Environment String.
DosGetVersion Returns the DOS version number.

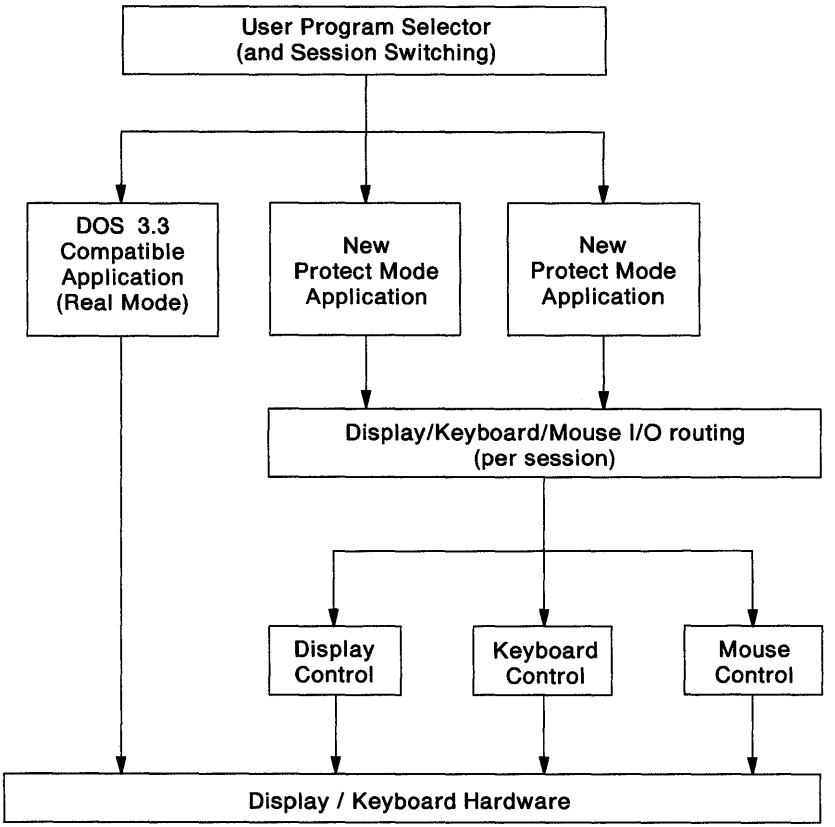
Chapter 5. Session Management

The top layer of the session manager is called the shell. The shell interfaces with the application user. The OS/2 shell is the Program Selector. The Program Selector allows the application user to start and switch among sessions. One DOS mode and multiple OS/2 mode sessions are supported.

The shell uses lower level session manager functions to start and switch among applications. Each application runs in its own session and has its own screen. One session is visible and is called the foreground session. The other sessions are called background sessions. Background OS/2 sessions continue to run until they issue some function which causes them to wait, for example, request keyboard input. The DOS mode is frozen in the background.

The video, keyboard, and mouse subsystems support the session manager. Screen, keyboard, and mouse I/O are routed on a per session basis. The session manager with the assistance of the video subsystem saves the contents of the screen over a screen switch. Application participation, reference "VIO Screen Save/Restore Operations" on page 6-20, is required to save the contents of the screen for graphics modes.

The following diagram shows the Program Selector and the display/keyboard/mouse routing features of the Session Manager.



Session Manager Application Support

The following session manager functions are available to OS/2 mode application programs:

1. Start a program in another session and include the started program in the switch list. The new session becomes a child session of the calling program's session. Child sessions may be manipulated by the parent session as follows:

   - Enable a parent session to switch one of its child sessions to the foreground.
   - Enable a parent session to set one of its child sessions selectable or non-selectable.

     This option affects selections made by the operator from the shell switch list. It does not affect selections made by the parent session.
   - Enable a parent session to bind one of its child sessions to itself so that, when the parent session is selected, the child session is brought to the foreground.

     This option affects selections made by the operator from the shell switch list. It does not affect selections made by the parent session.
   - Enable a parent session to terminate one or all of its child sessions.

   Note: Although a parent session/child session relationship exists, a parent process/child process relationship does *not* exist. Refer to the paragraph "Parent/Child Relationship:" in the Remarks section of DosStartSession in *Technical Reference, Vol. 2.*

2. Start a program in another session and include the started program in the switch list. No relationship is established between the new session and the calling program.

Restrictions

The session manager interfaces described in this section may not be issued under the following conditions:

1. By a process started by RUN= in CONFIG.SYS.
2. By a process started by the Detach command.
3. During a VIO popup (by a process which has issued VioPopUp and not yet issued VioEndPopUp).

A total of 12 sessions may be started, whether by the operator through the shell menu or by an application through the API described in this section.

Session manager API calls issued by background processes will be blocked during a hard error or VIO popup.

Session Management API Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The session management API function calls are summarized as follows:

DosSelectSession	Select Session
DosSetSession	Set Session Status
DosStartSession	Start a Session
DosStopSession	Stop Session

Chapter 6. I/O Services

OS/2 provides I/O access to the major character and block devices through function calls. Some devices are accessed through function calls specific to the device, such as the keyboard (KBD) and video I/O (VIO) calls. A device such as a disk is accessed using file system function calls. In addition, the file system API is used to access any named character device, such as LPT1 or COM1.

Many I/O function calls use a parameter called a *handle*. A handle is a 16-bit value that refers to a particular device or file.

ASCIIZ Strings

Several I/O function calls accept an ASCIIZ string (an ASCII string terminated by a byte of binary zeros) as input.

Country Considerations: ASCIIZ strings can be composed of mixed single- and double-byte characters, and can be used in the following cases:

- Filename and filename extension
- Path name
- Directory name

Filename Specification

The OS/2 standard filename consists of 1 to 8 bytes (characters), optionally followed by a dot and extension. The standard filename extension consists of 1 to 3 bytes (characters). This 8.3 limit on the format may be expanded in the future; programs should not parse filename strings.

Leading blanks are not allowed in the filename specification. ASCII characters less than 20H are illegal as well as the following characters:

```
< > + = ; , . " / \ [ ]
```

A period (.) or dot is the delimiter between filename and extension. The standard filename definition of 8.3 bytes means that name

formats longer than 8.3 received from OS/2 mode applications are not truncated to the 8.3 format and considered acceptable. Instead, they are classified as erroneous names.

Name formats received from DOS 3.3 applications running in the DOS mode will be truncated to the 8.3 format. The resulting name is considered acceptable.

Name formats received from new applications written to the Family API and running under the DOS mode are not truncated to the 8.3 format. Instead, they are classified as erroneous names. This insures that the Family API behaves the same whether running in the DOS mode or the OS/2 mode.

Verification of valid filename characters uses the Country Support CDIDS (Country Dependent Information Data Structures) for Double Byte Character Set (DBCS) environmental vector and for file system character names. Refer to Chapter 11, "Country Support Considerations" on page 11-1 for more information.

Country Considerations: If the 8th byte of a filename or 3rd byte of the extension is the first byte of a double-byte character, then the name or extension is not truncated to 7 bytes or 2 bytes respectively, but instead is reported as an error.

Filenames can be entirely single-byte, mixed single- and double-byte or entirely double-byte characters. The use of a double-byte character counts as two bytes. All double-byte characters can be used, with the exception of double-byte space.

Device Names

The operating system has reserved certain names for devices supported by the base device drivers. These device names are listed as follows:

COM1-COM3	First through third serial ports
CLOCK$	Clock.
CON	Console keyboard and screen
SCREEN$	Screen.
KBD$	Keyboard
LPT1 or PRN	First parallel printer
LPT2	Second parallel printer
LPT3	Third parallel printer
NUL	Nonexistent (dummy) device
POINTER$	Pointer draw device (Mouse screen support)
MOUSE$	Mouse

These names can be used in the DosOpen function call to OPEN the corresponding devices. Note that these reserved device names take precedence over filenames; the OPEN always checks for a device name *before* checking for a filename. The only exception is that COM1-COM3 are only reserved device names when the ASYNC (RS232-C) Device Driver is loaded. This means that a filename which matches a reserved device name can never be OPENed, because the device will be OPENed instead.

Code Page Support

OS/2 code page management reads keyboard input and writes display and printer output for concurrent multiple processes that input and output ASCII based character data encoded in different code pages.

A code page defines a character set by assigning each character to a location in a code page table. A character set is implemented by using a "character shape" table from which a character is selected for output by its associated display or printer. A character set is either downloaded to a device or ROM resident at the device.

The system accomplishes this by switching the required code page for a code page supported device prior to input or output.

The required code page is the current code page of the process at the time it opens a device or a specific code page selected by the process with a set code page API function. A character set can also be plugged in the device, such as the Quietwriter® III[1] printer.

In addition, the country APIs retrieve country and language dependent information in the current code page of the calling process or in a code page selected by the process.

Code Page Management

Code Page Management allows a user to select a code page for keyboard input and screen and printer output before running an application, a system command or utility in the OS/2 multitasking environment. This allows the user in a particular country such as England (code page 437) or Norway (code page 865) or language region such as Canadian-French (code page 863) to run with a code page that defines an ASCII-based character set containing characters used by that particular country or language.

OS/2 supports the following code pages:

437 U.S. IBM PC code page
850 Multilingual code page
860 Portuguese code page
863 Canadian-French code page
865 Nordic code page

OS/2 allows the user to prepare one code page or a combination of two code pages from the above list. Installable code page files include keyboard translate tables, display character sets, printer character sets, and country/language information for each code page supported.

The primary features of Code Page Management are:

- User commands (CODEPAGE and DEVINFO) in CONFIG.SYS for commanding system initialization to prepare selected code pages

---

[1] Quietwriter is a registered trademark of International Business Machines Corporation

and devices for code page switching at run time when in character text modes.

- An OS/2 user command (CHCP) to change the code page of its session (command process that executes CHCP) and any application that runs in that session. CHCP affects printer code pages only on print jobs that are opened subsequent to the command.
- API functions for an application to set and query code page for a process, keyboard input, video output, and spooled printer output.
- System code page switching of keyboard, display, printer/spooler, and country information for concurrent processes in different code pages.

One system country code, one system keyboard layout, and up to two system code pages can be configured for system use at run time.

Code Page Dependent Information

System information dependent on the code page includes:

- Video character set for display
- Keyboard translate table for scan code to character conversion
- Printer translate table and ROM code pages for printer
- Country format information for time, date, and other formats
- Language collate sequence table for character string sorting
- Language case map table from lower to upper case
- Language DBCS environment vector of lead bytes

This information is created at system initialization and maintained in system storage by OS/2 for each system code page that is prepared and is retrieved as needed according to code page.

Code Page Switching Examples

Some examples of code page switching that can occur are:

- The printer character set is switched by the Print Spooler to the process code page (that opens the printer) before it outputs the process write data stream to the printer.
- A process requests DosGetCtryInfo for the country information of the system country code or another selected country code and

the information is provided encoded in the code page of the process.

Switching the display to the code page that a system ASCII message is encoded in prior to output is not provided.

Data file code page tagging and code page switching based on the file code page tag is not provided. Also, filenames created under one code page may not be accessible by that name under another code page because the name characters may case map differently. This can be avoided by using only the first 128 characters of a code page for filenames or only unaccented characters (a-z, A-Z, and 0-9).

Code Page Preparation

During system initialization the selected code pages specified in the CODEPAGE command are prepared to allow run time code page switching of the display, the keyboard, the printer(s), and the country information. The display, keyboard, and printer(s) to be prepared must be defined in a DEVINFO command. Country information is prepared for the system country code specified in the COUNTRY command. If a resource cannot be prepared for the selected code page(s), then it is prepared for a default code page.

System resources default in the following ways at system initialization for code page preparation when the code page cannot be found for the resource.

- A keyboard layout defaults to the code page of the translate table designated as the default layout in the KEYBOARD.DCP file. The default layout is based on the national code page of its associated country. KEYBOARD.DCP must be explicitly specified in the DEVINFO statement for the keyboard in CONFIG.SYS.
- The display defaults to the code page of ROM_0 for the device.
- The printer defaults to the code page of ROM_0 for the device.
- The country information defaults to the code page of the first entry found in the COUNTRY.SYS file for the country code. Each entry is the same information for a given country code but encoded in a different code page. The first entry is based on the preferred country code page.

Note: ROM_0 means a device default code page which is the device native code page or the lowest addressed ROM code page.

In the following table, countries are shown on the left, country codes are shown next, their default code page assignment is listed in the third column, and country keyboard layouts are shown on the right. If country information cannot be prepared at system initialization because it is not found in the COUNTRY.SYS file for a code page selected with the CODEPAGE command, then it is prepared (maintained for run time code page switching in memory) in the default code page. Similarly, a keyboard layout is prepared in its default code page if it cannot be prepared in the selected code page because it is not found in the KEYBOARD.DCP file. COUNTRY.SYS contains one default entry per country code and KEYBOARD.DCP contains one default entry per keyboard layout based on these assignments:

Country	Ctry Code	Code Page	Kbd
Asia	099	437	-
Australia	061	437	-
Belgium	032	437	BE
Canada	002	863	CF
Denmark	045	865	DK
Finland	358	437	SU
France	033	437	FR
Germany	049	437	GR
Italy	039	437	IT
Latin America	003	437	LA
Netherlands	031	437	NL
Norway	047	865	NO
Portugal	351	860	PO
Spain	034	437	SP
Sweden	046	437	SV

Country	Ctry Code	Code Page	Kbd
Switzerland	041	437	SF,SG
United Kingdom	044	437	UK
United States	001	437	US

Although only up to two code pages can be selected with the CODEPAGE command, the system may actually have prepared three or more code pages at system initialization in case a system resource defaults on code page preparation. For example, the keyboard may be prepared for a default code page that is different than the two selected code pages for which the display is successfully prepared.

Code Page Operation

Each process has a code page tag maintained by the OS/2 kernel. A code page is a table that defines how characters are encoded. A code page tag is the identifier of the current code page for the process. See the *IBM Operating System/2™ Programmer's Guide* for a description of code pages and the *IBM Operating System/2™ User's Reference* for information on configuring code pages for the system.

A child process inherits the code page tag value of its parent. The default code page for the first process of a program started in a session is the same as the session code page. The default code page for a new session is the primary code page specified in the CODEPAGE configuration command. A process code page tag may be changed with the function call DosSetCp or DosSetProcCp and queried by DosGetCp. However, DosSetCp or DosSetProcCp does not change its parent or any child process code page tags. See *Technical Reference, Vol. 2* for details about these function calls.

The following explains what code page is used when performing process input and output:

- Spooled printer output by a process is printed by the spooler in the code page of the process. The code page that the spooler uses to print is established through the system spooler and file system at the time the process makes an open printer request.
- Video output by a process is shown by the Video Subsystem in the current code page of the implied video handle being used for output by the process. The default display code page for a new session is the primary code page specified in the CODEPAGE configuration command or defaults to the display ROM code page. The display code page can be changed by a process with VioSetCp or DosSetCp for the logical display of the session to which the calling process belongs.
- Keyboard scan code input is converted into ASCII characters by the keyboard device driver in the current code page of the keyboard handle being used for input by a process. The default keyboard handle code page for a session is the primary code page specified in the CODEPAGE configuration command or defaults to US437. The keyboard code page can be changed by a process with KbdSetCp or DosSetCp for the logical keyboard of the session to which the calling process belongs.
- Utility and command output is shown in the current code page of the command process which executes the function.

An invocation of a CMD.EXE process in OS/2 or a COMMAND.COM process in the DOS mode is a session. Therefore, the code page of the command process is the session code page. The user command CHCP is provided to change the code page of a session. CHCP performs the following functions:

- Sets the session's command process code page
- Sets the session's logical display code page
- Sets the session's logical keyboard code page.

The printer code page is based on the process code page at the time the process makes an open printer request. No special command is needed for the printer.

Code Page Supported Devices

Device code page support is provided for the keyboard and certain printer and display devices with code page download and switch capability.

The devices supported by OS/2 code page switching include:

- IBM Personal System/2™ Display Adapter
- IBM Proprinter™[2]
- Quietwriter® III[3]
- Enhanced Graphics Adapter (EGA)
- Video Graphics Array (VGA)

Special Considerations and Limitations

Code Page Management has these special considerations and limitations:

- One system country code is prepared at system initialization and cannot be changed at run time; however, information for other country codes can be retrieved through the Country API for country information.
- One or two system code pages may be prepared at system initialization. The user can switch (CHCP) between them and so can a program through the API described in the following section. The system code pages are prepared based on CONFIG.SYS commands and system initialization and cannot be changed at run time.
- One system keyboard layout is prepared at system initialization and can be changed at run time using the KEYB utility. The new selected keyboard layout replaces the current keyboard translate tables for the system code pages defined by the CODEPAGE command in CONFIG.SYS, if the selected layout is available in those code pages.

---

[2] Proprinter is a trademark of International Business Machines Corporation

[3] Quietwriter is a registered trademark of International Business Machines Corporation

- The CODEPAGE command allows preparation for up to two selected code pages. However the system may actually have more than two different code pages active in the system in these cases:
  - System Initialization defaults to a code page different than the selected code pages for a printer character set, display character set, keyboard layout or the country information when the selected code page is not available.
  - KEYB defaults to a different code page for the keyboard layout if it is not available in a selected code page.
  - An application uses printer IOCtl to activate additional printer code pages.
- The CHCP command only changes the code page of the command process to which it belongs and does not affect the code page of any other existing command process, other process or other subsystem handle. Its purpose is to allow the user to change the code page of the session.
- The user can switch between two prepared code pages with the CHCP command and the application can switch between code pages with the appropriate APIs. The primary (default) system code page is used throughout the system and by all applications if a switch never occurs following system initialization. If no system code pages have been prepared, then the system runs with the default code page available per device, country information, and keyboard layout.
- An application controls the appearance of its screen display characteristics when using OS/2 APIs to switch code pages. Characters may not appear the same for different code pages. The application can offer code page selection to the user, if required, through its own application user interface.
- The OS/2 system spooler must be installed for printer code page switching or be replaced by another printer monitor that receives code page commands from the printer device driver and provides the code page function support.
- It is possible for an application to issue printer IOCtl commands to activate different code pages throughout its printed data stream. However, no data stream format is defined by OS/2. Imbedded escape sequences and printer control codes are allowed and not monitored by OS/2.

- OS/2 does not automatically switch to the code page of a filename and does not keep track of the code page of a filename. This allows the following possible situations to occur:
  - The file is not accessible under a different code page because the filename is not recognized in that code page although valid in the original code page in which it was created.
  - The wrong file becomes accessible under a different code page because the keyboard entered filename maps to a different but valid filename in that code page.
- The system does not automatically switch to the code page of a message prior to output. Therefore, messages encoded in a particular code page may not be fully readable in another code page.

Code Page API Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The code page function calls are summarized as follows:

DosSetCp	Set the code page of the calling process and the session's display and keyboard code page
DosSetProcCp	Set the code page of the calling process
DosGetCp	Get the code page of the calling process and the system code page(s)
VioSetCp	Set a video subsystem code page
VioGetCp	Get a video subsystem code page
KbdSetCp	Set a keyboard subsystem code page
KbdGetCp	Get a keyboard subsystem code page
Printer IOCtl	Requires Print Spooler to be installed • Activate a printer code page • Get the active printer code page • Validate a printer code page

System Initialization

OS/2 runs on the Personal Computer AT® model group, Personal Computer XT™ Model 286 model group, and the PS/2 model group. This is supported by loading the correct set of device drivers by model group. A different set of reserved filenames will contain the base device drivers for each supported model group that are not loaded by CONFIG.SYS processing. The kernel adapts itself to the correct model in the hardware-dependent areas that are not device driver related.

Hardware Characteristics

OS/2 assumes that the following characteristics of PS/2 devices are statically assigned before OS/2 initialization:

- "Sleep" or "wake" status
- Interrupt level
- I/O port(s)

These device characteristics may not change after OS/2 initialization has begun.

Device Driver Installation

OS/2 automatically loads all base device drivers, which are model group dependent. OS/2 and the system installation procedure are not dependent on the specification of any device drivers in the CONFIG.SYS file for normal system operation.

The system install process inserts the appropriate DEVICE = statements for all user specified device drivers into the CONFIG.SYS file. The names of these device drivers are based on the type of device driver the user has specified and the model group on which the system install process is running. The system install process also copies the correct device driver file to the IPL volume.

If the user wants to manually insert DEVICE = statements into the CONFIG.SYS file, the user needs to ensure that the device driver being installed is correct for the model group the system is started on.

The CONFIG.SYS file must not contain DEVICE= statements for the base device drivers required for basic OS/2 operation.

Device I/O Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The device I/O function calls are summarized as follows:

DosBeep	Generate sound from speaker
DosDevConfig	Get device configuration
DosDevIOCtl	I/O control for devices
DosPortAccess	Request I/O access to devices
DosPhysicalDisk	Partitionable Disk support
DosCLIAccess	Request CLI/STI Access

File I/O Services

File handle values of FFFFH do not represent actual file handles but are used throughout the file system interface to indicate specific actions to be taken by the file system. Usage of this *special file handle* where it is not expected by the file system results in an error.

Note: Null pointers are defined to be 00000000H throughout this book.

Existing file systems that conform to the Standard Application Program Interface (Standard API) described in this section, do not necessarily support all the described information kept on a file basis. When such is the case, file system drivers return to the application a null (zero) value for the unsupported parameter.

Note: The order of processing of multiple outstanding requests issued by multiple threads is not guaranteed. See "Request Packets" on page 7-37 for a discussion of the order in which requests are issued to the API by multiple threads and the order in which the requests arrive at a device driver.

File I/O Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The file I/O function calls are summarized as follows:

DosBufReset	Flush file buffers
DosChDir	Change current directory
DosChgFilePtr	Change (Move) the file read or write pointer
DosClose	Close file handle
DosDelete	Delete file
DosDupHandle	Duplicate file handle
DosFileLocks	Lock or unlock multiple ranges of bytes in an opened file
DosFindClose	Terminate usage of a directory search handle
DosFindFirst	Find first matching file
DosFindNext	Find next matching file
DosMkDir	Make subdirectory
DosMove	Move a file
DosNewSize	Change size of a file
DosOpen	Open or create a file
DosQCurDir	Query current directory
DosQCurDisk	Query current default drive
DosQFHandState	Query file handle state
DosQFileInfo	Query file information
DosQFileMode	Query file mode
DosQFsInfo	Query file system information
DosQHandType	Query handle type
DosQVerify	Query the verify setting
DosRead	Read from a file
DosReadAsync	Asynchronous Read from a file
DosRmDir	Remove subdirectory
DosScanEnv	Scan environment segment
DosSearchPath	Search a path for a filename
DosSelectDisk	Select disk
DosSetFileInfo	Set file information
DosSetFileMode	Set file mode
DosSetFHandState	Set file state
DosSetFsInfo	Set file system information
DosSetMaxFH	Define new maximum file handle
DosSetVerify	Set verify setting
DosWrite	Write to a file or device
DosWriteAsync	Asynchronous write to a file or device

Video I/O Services

Display Adapters Supported

OS/2 supports the following adapters:

- Color Graphics Adapter (CGA) - Personal Computer AT® or Personal Computer XT™ Model 286
- Enhanced Graphics Adapter (EGA) - Personal Computer AT or Personal Computer XT Model 286
- Video Graphics Array (VGA) - PS/2 only
- IBM Personal System/2™ Display Adapter - Personal Computer AT or Personal Computer XT Model 286
- IBM Personal System/2™ Display Adapter 8514/A - PS/2 only

Video I/O (VIO) services are provided for any one of these adapters. Configurations including multiple displays are not supported.

Video Graphics Array (VGA)

The VGA is EGA compatible with the exception that it has additional video modes.

Differences between VGA and EGA include:

- EGA does not run on the PS/2.
- VGA runs ONLY on the PS/2.
- VGA supports additional text and graphics modes
- VGA starts in a different mode than the EGA display. When a PS/2 monochrome display is present, the VGA starts in high resolution monochrome text mode (mode 7+, 80x25 alphanumeric text support with a 9x16 character cell and a 720x400 pixel resolution). When a PS/2 color display is present, the VGA starts in high resolution color text mode (mode 3+, 80x25 alphanumeric text support with a 9x16 character cell and a 720x400 pixel resolution). The OS/2 software starts with this display set to normal video mode (white letters on a black background).

IBM Personal System/2™ Display Adapter

OS/2 supports the IBM Personal System/2™ Display Adapter and the IBM Personal System/2™ Display Adapter 8514/A. The IBM Personal System/2™ Display Adapter is VGA-compatible and works on the Personal Computer AT and on the Personal Computer XT Model 286. VGA-compatible modes are supported on the IBM Personal System/2™ Display Adapter.

The IBM Personal System/2™ Display Adapter 8514/A is supported on the PS/2. VGA-compatible modes are supported on this adapter.

VIO Support by Mode

The following modes are supported by VIO:

Text - Modes 0, 1, 2, 3 and their + and * variations, mode 7 and the + variation.

Graphics - All Points Addressable (APA) modes 4, 5, 6, D, E, F, 10, 11, 12, and 13.

Note: Only a subset of VIO calls are supported in graphics modes. See "VIO Calls Supported in Graphics Modes" on page 6-19 for details.

Not all modes are supported on all adapters.

Text Modes Supported (Mono-Compatible)

A/N represents Alpha/Numeric in the following charts.

The following Text modes are supported by the EGA, VGA, and IBM Personal System/2™ Display Adapter:

Mode	Type	Text	Colors	Cell	Resolution
7	A/N	80x25	--	9x14	720 x 350
7+	A/N	80x25	--	9x16	720 x 400

Note: The '+' mode variation is only supported by the VGA and IBM Personal System/2™ Display Adapter.

The following Text modes are supported by the CGA, EGA, VGA, and IBM Personal System/2™ Display Adapter:

Mode	Type	Text	Colors	Cell	Resolution
0, 1	A/N	40x25	16	8x8	320 x 200
0*, 1*	A/N	40x25	16	8x14	320 x 350
0+, 1+	A/N	40x25	16	9x16	360 x 400
2, 3	A/N	80x25	16	8x8	640 x 200
2*, 3*	A/N	80x25	16	8x14	640 x 350
2+, 3+	A/N	80x25	16	9x16	720 x 400

Note: The '*' variations on the above modes are only supported by the EGA, VGA, and IBM Personal System/2™ Display Adapter. The '+' variations on the above modes are only supported by the VGA and IBM Personal System/2™ Display Adapter. For modes 0 and 2, the color burst is turned off on the CGA.

VIO Calls Supported in Text Modes:

All VIO calls are supported in text modes.

Graphics Modes Supported

The following graphics All Points Addressable (APA) modes are supported by the CGA, EGA, VGA, and IBM Personal System/2™ Display Adapter:

Mode	Type	Text	Colors	Cell	Resolution
4, 5	APA	40x25	4	8x8	320 x 200
6	APA	80x25	2	8x8	640 x 200

Note: For modes 5 and 6, the color burst is turned off on the CGA.

The following additional APA modes are supported by the EGA, VGA, and IBM Personal System/2™ Display Adapter:

Mode	Type	Text	Colors	Cell	Resolution
D	APA	40x25	16	8x8	320 x 200
E	APA	80x25	16	8x8	640 x 200
F	APA	80x25	--	8x14	640 x 350
10	APA	80x25	16*	8x14	640 x 350

* Only 4 colors are available on an EGA configuration with less than 128Kb of video memory.

The following additional APA modes are supported by the VGA and IBM Personal System/2™ Display Adapter:

Mode	Type	Text	Colors	Cell	Resolution
11	APA	80x30	2	8x16	640 x 480
12	APA	80x30	16	8x16	640 x 480
13	APA	40x25	256	8x8	320 x 200

VIO Calls Supported in Graphics Modes

The VIO calls supported in graphics modes are a subset of those VIO calls supported for text modes. The following calls are supported:

VioDeRegister Deregister a video subsystem
VioEndPopUp Deallocate a popup display screen
VioGetConfig Get Video Configuration
VioGetFont Get Font (request type 1 only)
VioGetMode Get display mode
VioGetPhysBuf Get physical display buffer
VioGetState Get Video State (request types 0 and 1 only)
VioModeUndo Undo previous restore mode registration
VioModeWait Wait for restore mode notification
VioPopUp Allocate a popup display screen
VioRegister Register a video subsystem
VioSavRedrawWait Wait for screen save/redraw notification

VioSavRedrawUndo	Undo previous save/redraw registration
VioScrLock	Lock screen for I/O
VioScrUnLock	Unlock screen for I/O
VioSetMode	Set display mode
VioSetState	Set Video State (request types 0 and 1 only)

The other VIO calls, Print Screen, and Control Print Screen are not supported in graphics modes.

VIO Screen Save/Restore Operations

Screen save/restore operations are triggered in OS/2 by any of the following events:

1. When the operator uses the hotkey to switch to another application,
2. When an application issues DosStartSession (specifying foreground),
3. When an application issues DosSelectSession,
4. When an application issues VioPopUp (or a hard error popup is displayed), and
5. When an application issues VioEndPopUp (or a hard error popup ends).

For graphics modes, OS/2 notifies the application to perform the required save/restore operation. To be notified for events 1 through 3 above, a graphics mode application issues VioSavRedrawWait. The return from the VioSavRedrawWait call provides the notification. A parameter returned on the call tells the application whether to perform a save or restore. To be notified for event 5, the application issues VioModeWait, and the return from VioModeWait is the notification. There is no notification for event 4.

When an application's VioSavRedrawWait thread is notified to perform a save, the application must save the physical display buffer, video mode, state, and any other display adapter registers the application may have modified.

When an application's VioSavRedrawWait thread is notified to perform a restore, it must restore the physical display buffer, video mode, state, and modified display adapter registers. When an appli-

cation's VioModeWait thread is notified to perform a restore, the application must restore the video mode, state, and modified display adapter registers. An application's VioModeWait thread does not restore the physical display buffer. OS/2 saves/restores the physical display buffer over a popup.

Note that a screen switch may occur while the foreground application is currently accessing (under the protection of VioScrLock) the physical display buffer. In this case, the screen switch remains pending until either the application issues VioScrUnLock or the screen lock times out.

A graphics mode application must issue VioSavRedrawWait. A graphics mode application must issue VioModeWait only if it writes directly to the registers on the display adapter. If VioModeWait is not issued, OS/2 will restore the physical display buffer, mode, and state at the completion of a popup.

Screen save and restore operations for text modes are performed automatically by OS/2. A text mode application must issue VioSavRedrawWait and VioModeWait only if it writes directly to the registers on the display adapter.

An application's VioSavRedrawWait thread may be notified to perform a restore before it is notified to perform a save. This would happen if the application was running in the background when it first issued VioSavRedrawWait.

Note: The OS/2 Start command starts an application in the background.

VIO Code Page Support

VIO code page support is provided for a subset of the display adapters supported in OS/2. VIO code page support is provided only for those adapters whose hardware supports down loadable fonts. The selection is limited to the following adapters:

- EGA (Enhanced Graphics Adapter) - Personal Computer AT and Personal Computer XT Model 286
- VGA (Video Graphics Array) - IBM Personal System/2™ only
- IBM Personal System/2™ Display Adapter - Personal Computer AT® and Personal Computer XT™ Model 286

- IBM Personal System/2™ Display Adapter 8514/A - IBM Personal System/2™ only

VIO allows an application to select one of two user-specified code pages as the current video code page. Alternately, the application may specify code page 0000 which corresponds to the ROM-resident code page. The following two VIO calls support code pages:

- VioSetCp - sets the current video code page
- VioGetCp - returns the current video code page

VioSetCp can be used to specify one of the two code pages defined on the CODEPAGE statement in CONFIG.SYS.

Code page 0000, the ROM-resident code page, may also be specified on VioSetCp. Code page 0000 is also used as the current video code page in the following cases:

- No primary code page is identified during system initialization.
- A primary code page is identified, but the corresponding video code page file is not found. This case will occur if there is no DEVINFO statement identifying a video code page file in CONFIG.SYS, or if the video code page file identified is not found.

Code page 0000 may not be specified on the CODEPAGE= statement in CONFIG.SYS.

VioSetCp associates the code page specified on the call with the VIO handle also specified. Because handle zero is the only VIO handle supported, all threads and processes within a session share the same handle and current video code page.

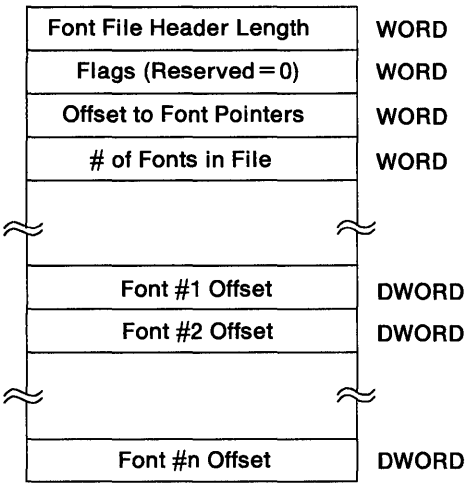
Applications may not manipulate the Character Map Select Register.

Video Font File Organization

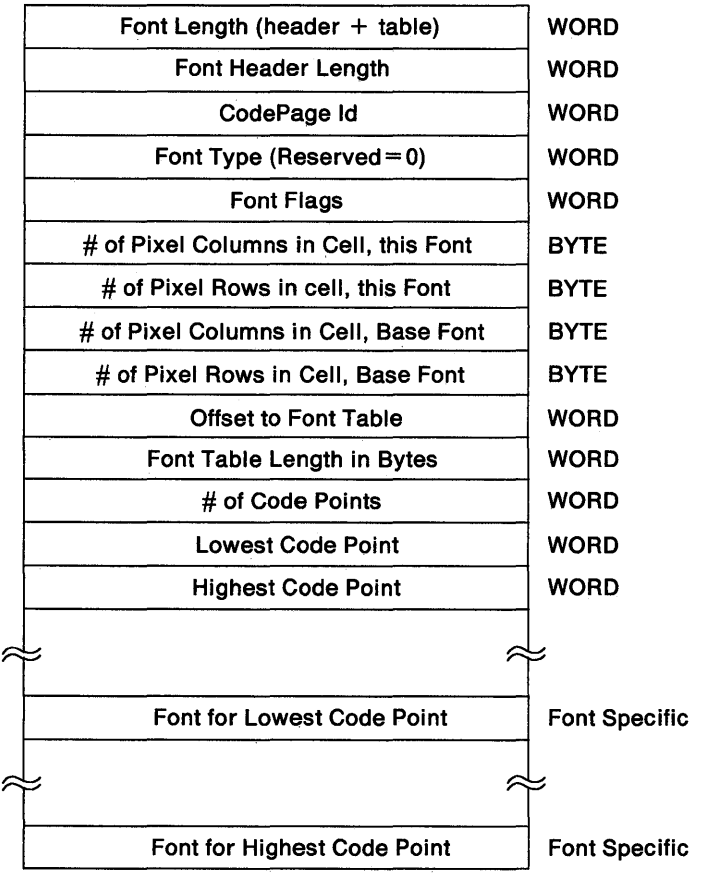
There is one video font file which contains all the fonts for all the code pages supported. Fonts are extracted from the video font file by the video subsystem.

The video font file contains a file header and multiple font tables.

Video Font File Header: The font file header is shown below:



Video Font Table Format: Each font table has the following format:



The Font Flags field has the following values:

Bit 0=1 Partial font.
Bit 1=1 Code points are included with each font.
Bit 2-15 Reserved=0

Additional VIO Considerations

- The default mode used in the OS/2 mode is the highest resolution supported by the primary display. Any conflicting switch settings on the adapter are ignored.
- VIO calls issued by multiple threads and processes within a session are serialized via a semaphore. Applications with multiple threads and processes issuing VIO calls within a session should be aware of the following potential lockout situations:
  - Issuing a VIO call within a critical section. This action will fail if the thread entering the critical section gains control at a time when another thread in the same process is in the middle of a VIO call. Reference DosEnterCritSec and DosExitCritSec in *Technical Reference, Vol. 2* for more information.
  - Allowing a popup process to request a semaphore (or resource) owned by another process within the same session. This action will fail if the process owning the resource gets blocked on a VIO call.

VIO Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The VIO function calls are summarized as follows:

VioDeRegister	Deregister video subsystem
VioEndPopUp	Deallocate a popup display screen
VioGetAnsi	Get ANSI status
VioGetBuf	Get logical video buffer
VioGetConfig	Get video configuration
VioGetCp	Get code page
VioGetCurPos	Get cursor position
VioGetCurType	Get cursor type
VioGetFont	Get font
VioGetMode	Get display mode
VioGetPhysBuf	Get physical display buffer
VioGetState	Get video state
VioModeUndo	Undo previous restore mode registration
VioModeWait	Wait for restore mode notification
VioPopUp	Allocate a popup display screen
VioPrtSc	Print screen

VioPrtScToggle	Ctrl-PrtSc notification
VioReadCellStr	Read character attribute string
VioRegister	Register video subsystem
VioReadCharStr	Read character string
VioSavRedrawWait	Wait for screen save/redraw notification
VioSavRedrawUndo	Undo previous save/redraw registration
VioScrLock	Lock screen for I/O
VioScrUnLock	Unlock screen for I/O
VioScrollDn	Scroll screen down
VioScrollLf	Scroll screen left
VioScrollRt	Scroll screen right
VioScrollUp	Scroll screen up
VioSetAnsi	Turn ANSI on or off
VioSetCp	Set code page
VioSetCurPos	Set cursor position
VioSetCurType	Set cursor type
VioSetFont	Set font
VioSetMode	Set display mode
VioSetState	Set video state
VioShowBuf	Update display with logical video buffer
VioWrtCellStr	Write character attribute string
VioWrtCharStr	Write character string
VioWrtCharStrAtt	Write character string with attribute
VioWrtNAttr	Replicate attribute
VioWrtNCell	Replicate cell
VioWrtNChar	Replicate character
VioWrtTTY	Write TTY string

DOS Mode EGA Considerations

For some DOS mode EGA applications, OS/2 will not be able to switch from DOS mode to OS/2 mode and then back again. Upon return to the DOS mode application, the screen will be incorrect. The DOS mode EGA applications that do not run successfully are:

- Applications that download fonts into a character generator block other than block 0; Character generator block 0 is supported. (See EGA ROM BIOS function call INT 10H, AH = 11H.)
- Graphics mode applications that use more than one display page.
- Advanced graphics mode applications that write directly to the registers on the EGA adapter.

To supplement OS/2 screen switching support, a DOS mode application can be written to use the EGA Register Interface. See "EGA Register Interface" on page 9-102. Alternately, a DOS mode application can be notified on a screen switch via Multiplex Interrupt 2FH, AH=40H.

Note: On an IBM Personal System/2™ or an IBM Personal Computer AT with the IBM Personal System/2™ Display Adapter, the registers on the adapter are both readable and writable. For these configurations, OS/2 reads and saves the registers on a screen switch away from DOS mode and restores the registers upon return to DOS mode.

OS/2 issues a new Multiplex Interrupt (INT 2FH) to signal the following two events: moving the DOS mode application to the background (AX=4001H) and moving the DOS mode application to the foreground (AX=4002H). A DOS mode application that wishes to receive this signal must "hook" the Multiplex Interrupt vector. That is, when the application is started, it must save the current INT 2FH vector and set the INT 2FH vector to point to the application's own interrupt handler.

When the notification is received, the application must save all registers, perform whatever processing is required, restore all registers, and issue the IRET instruction to return to OS/2. Only the following forms of processing are supported:

- Modifying application and/or video memory,
- Issuing ROM BIOS video service calls (INT 10H),
- Issuing EGA Register Interface calls (INT 10H), and
- Programming the EGA video card directly.

Note: If an application moving to the background uses the EGA Register Interface to save the EGA registers, these registers are restored automatically when the application is returned to the foreground.

An application may receive notification that it is moving to background at any time. At the point this notification occurs, the application (other than its INT 2FH handler) is frozen until it is returned to the foreground. Code sequences that are sensitive to interruption can be protected with CLI/STI.

When an application's INT 2FH handler receives notification with a value in AH other than 40H, the application must issue the JMP FAR instruction to branch to the previous INT 2FH vector.

Keyboard I/O Services

Keyboard I/O Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The keyboard I/O function calls are summarized as follows:

KbdCharIn	Read character, scan code
KbdClose	Close a logical keyboard
KbdDeRegister	Deregister keyboard subsystem
KbdFlushBuffer	Clear the keystroke buffer
KbdFreeFocus	Free the previous physical to logical bind
KbdGetCp	Get loaded translate table IDs
KbdGetFocus	Bind the physical keyboard to a logical keyboard
KbdGetStatus	Get keyboard status
KbdOpen	Open a logical keyboard
KbdPeek	Peek at a character
KbdRegister	Register keyboard subsystem
KbdSetCp	Set the translate table
KbdSetCustXt	Install custom translate table
KbdSetFgnd	Set foreground keyboard priority
KbdSetStatus	Set keyboard status
KbdStringIn	Read character string
KbdSynch	Synchronize keyboard access
KbdXlate	Translate scan code

Binary Versus ASCII I/O

A user process performs I/O to a character device in either binary or ASCII modes. These modes are set by the user process through the IOCtl facility. In binary mode, data is transferred exactly as it appears and for the data length that the user requested. In ASCII mode, data can be edited, and/or translated by OS/2. The operations that OS/2 performs for ASCII mode I/O are listed below.

For a read in ASCII mode:

- A caret (^) is a symbol meaning: *press the Ctrl key*. The following characters preceded with a caret (^): ^C, ^Break, ^S, ^P, and ^PrtSc are handled specially.
- The data is read until the first ^M or ENTER key is seen. This means that the length of the read data can be less than the requested length. Note that the data is always terminated with the byte sequence ODH OAH.
- If ^Z is encountered, no further data is read.
- Data will echo to the standard output device (screen) only if echo mode is ON.
- Tabs are expanded into 8-character boundary spaces upon echo, but left as 09H in the buffer.

For an ASCII mode write:

- The ^S is interpreted for flow control.
- The ^P or ^PrtSc toggles printer echoing.
- The ^C or ^Break generates a signal for control-break handling.

  Note: The type ahead buffer is flushed for control-break but not for ^C.
- Tabs are expanded to 8-character boundaries and filled with spaces.
- ASCII character codes less than 20H are preceded with a caret (^) and 40H is added to the codes.

  ^T and ^U are not included in order to support certain foreign currency symbols.
- Output is performed up to (but not including) a ^Z or the length of the request. The number actually written can be less than the number requested.

A user process performs I/O to a block device strictly in binary mode. Data is transferred without interpretation or translation.

Mouse I/O Services

Mouse I/O Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The mouse I/O function calls are summarized as follows:

MouClose	Closes the mouse device for the current session.
MouDeRegister	Deregister mouse subsystem
MouDrawPtr	Release screen area for device driver use
MouFlushQue	Flush mouse event queue
MouGetDevStatus	Query current mouse device driver status flags
MouGetEventMask	Query current mouse device 1 word event mask
MouGetNumButtons	Query number of buttons
MouGetNumMickeys	Query number of mickeys per centimeter
MouGetNumQueEl	Query current status for the mouse device event queue
MouGetPtrPos	Query current pointer position
MouGetPtrShape	Query pointer shape and size
MouGetScaleFact	Query scale factors for the current mouse device
MouInitReal	Initialize DOS Mode pointer draw
MouOpen	Opens the mouse device for the current session
MouReadEventQue	Read the mouse device event queue
MouRegister	Register mouse subsystem
MouRemovePtr	Reserve screen area for application use
MouSetDevStatus	Set mouse device driver status flags
MouSetEventMask	Assign new event mask to the current mouse device
MouSetPtrPos	Set current pointer position
MouSetPtrShape	Set pointer shape and size
MouSetScaleFact	Set scale factors for the current mouse device
MouSynch	Get synchronous access

DOS Mode INT 33H Mouse API

OS/2 supports a subset of the Microsoft®[4] DOS INT 33H mouse API.

The Microsoft INT 33H mouse API is available only to those applications executing in the DOS mode. OS/2 mode applications must use the mouse API device interface.

See "DOS Mode INT 33H Mouse API" on page 9-61 for a detailed description of the DOS mode INT 33H function calls.

The DOS mode INT 33H Mouse API I/O function calls are summarized as follows:

INT 33H - Function 0 Mouse installed flag and reset
INT 33H - Function 1 Show mouse pointer
INT 33H - Function 2 Hide mouse pointer
INT 33H - Function 3 Get mouse pointer position and button status
INT 33H - Function 4 Set mouse pointer position
INT 33H - Function 5 Get button press information
INT 33H - Function 6 Get button release information
INT 33H - Function 7 Set minimum and maximum horizontal position
INT 33H - Function 8 Set minimum and maximum vertical position
INT 33H - Function 9 Set graphics pointer shape
INT 33H - Function 10 Set text pointer shape
INT 33H - Function 11 Read mouse motion counters
INT 33H - Function 12 Set user-defined subroutine input mask
INT 33H - Function 13 Light pen emulation on
INT 33H - Function 14 Light pen emulation off
INT 33H - Function 15 Set mickey/pixel ratio
INT 33H - Function 16 Conditional off
INT 33H - Function 19 Set double speed threshold
INT 33H - Function 20 Swap user-defined subroutine
INT 33H - Function 21 Query save mouse state storage requirements
INT 33H - Function 22 Save mouse driver state
INT 33H - Function 23 Restore mouse driver state

---

[4] Microsoft is a registered trademark of Microsoft Corporation

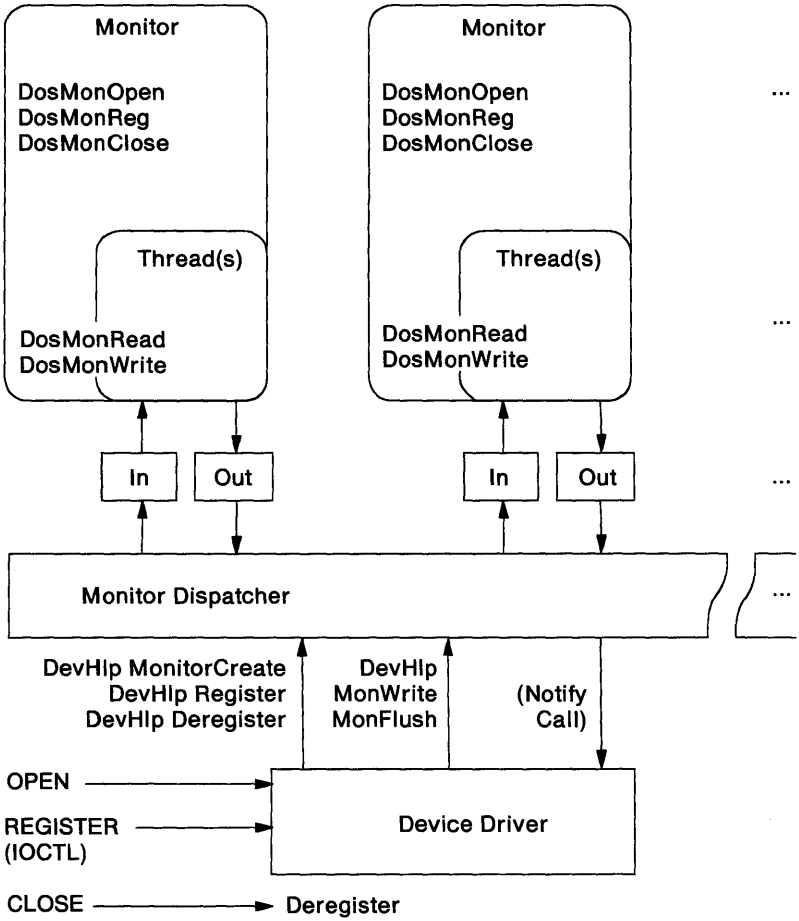
Device Monitor Services

Character Device Monitors

Character Device Monitors provide a mechanism for applications or subsystems to monitor all characters passing through a device driver. This mechanism allows any registered process to remove, insert or modify the information passing through the device.

The OS/2 monitor mechanism consists of the OS/2 monitor calls and the monitor dispatcher device helper. The monitor calls provide the interface for the monitor to interact with the device driver and the monitor's own input/output data buffers. The monitor dispatcher device helper handles the interfaces for the device driver and the mechanics of passing data from the output buffer of one monitor to the input buffer of the next monitor in the chain.

Monitor Details



Monitor Processes

A character device monitor is an application process, or part of an application process; that is, it runs at protection level 3 using standard OS/2 function calls (DosMonXXX) to interact with the device driver and the monitor's own input and output data buffers.

For a process to gain access to a device driver's data stream for character monitoring, the monitor must do the following:

1. Issue the DosMonOpen call to establish a connection to the device driver for monitors. DosMonOpen gets a device handle that will be used in subsequent DosMonReg and DosMonClose calls.
2. Issue the DosMonReg call to register a pair of input and output buffers with the device driver and monitor dispatcher. After the monitor is installed in the monitor chain, the monitor dispatcher automatically moves data between monitors (if there is more than one in the chain) and between the last monitor in the chain and the device driver's monitor chain buffer.

The device handle for monitors returned from a DosMonOpen call is unique to that process. A process needs to call DosMonOpen only once per character device whose data stream(s) it wishes to monitor. If a process issues more than one DosMonOpen call to the same device, the same device handle will be returned. A process may then register one or more monitors on the same device handle for monitors.

Note: Until the monitor returns successfully from the DosMonReg call, no characters will enter the monitor's input buffer. It is the application's responsibility to synchronize completion of the DosMonReg call and the subsequent monitoring of the data stream with device input into the data stream. See the diagrams on pages 6-51 and 6-52 for examples.

After a monitor has gained access to the data stream, it may remove, insert, modify or view all characters in data records passing through the data stream by:

1. Issuing the DosMonRead call to move a data record from its input buffer to a private data area that the monitor process can access freely, and
2. Issuing the DosMonWrite call to move a data record from the monitor process private buffer into its output buffer.

A data record is defined minimally as a WORD containing flags meaningful to the monitors and devices whose data stream they are monitoring. The flag WORD is always the first WORD in the data record. A monitor can modify the data record received on the last DosMonRead call before calling DosMonWrite to return it to the device's data stream. However, because the character device driver is sensitive to these flags and to the entire data record, the monitor application should never alter this order within a data record and must conform to device driver "rules" on filtering data records. Refer to Chapter 9, Device Drivers for device-specific descriptions of flag WORDs and data records.

Data movement from the monitor's input buffer (DosMonRead) and into the monitor's output buffer (DosMonWrite) is synchronized with the device driver and monitor dispatcher. When a data record is moved into the monitor's input buffer, the monitor process is signaled by the monitor dispatcher that a data record is available for DosMonRead. When DosMonRead completes movement of one data record, the monitor dispatcher is signalled that a data record has been removed and there is now more room for new data records. The opposite is true for DosMonWrite.

Note: Monitors get control of all data in a data stream before an application can read it. Therefore, a monitor's processing must be fast, performing no I/O or semaphore waits. For example, there is no recovery if a monitor application does a read for a character from the device driver API buffer when it has not yet moved the character through its input and output buffers into the device driver monitor chain buffer. Complex processing in a monitor should be performed within a monitor thread separate from that which makes DosMonRead and DosMonWrite calls.

When a process no longer wishes to monitor a character device's data stream, it relinquishes its access to the data streams by calling DosMonClose. DosMonClose is issued with a specific opened device handle for monitors, as returned from a previous DosMonOpen. All monitors for the current process that registered using this handle (those monitoring the particular character device) are terminated.

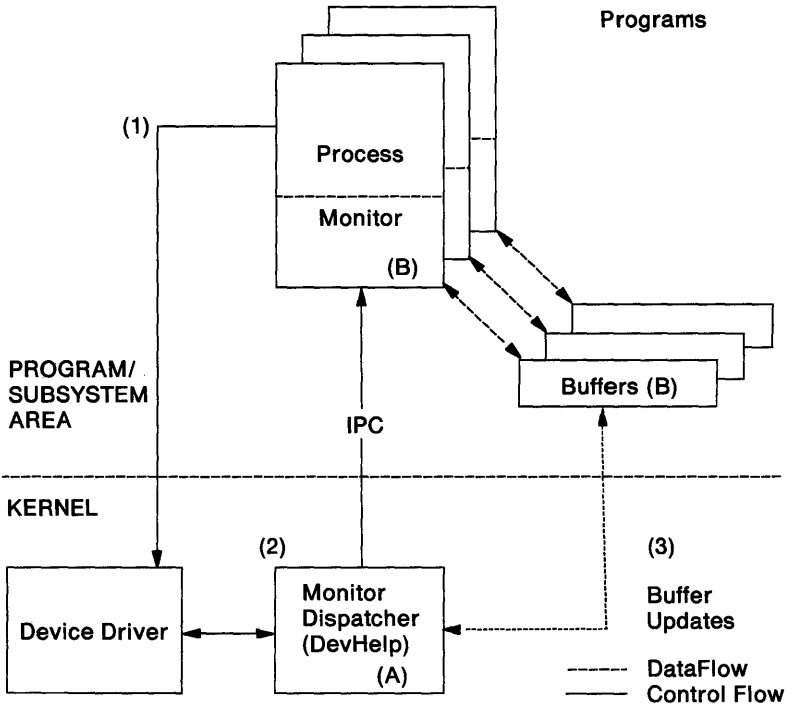
Pseudo code for a simple keystroke monitor is:

```
;****************************************
; SIMPLE CHARACTER DEVICE MONITOR
;****************************************

CALL DosSetPrty      ;Set this monitor's thread
                     ; priority high

; GET ACCESS TO THE DEVICE'S DATA STREAM

CALL DosMonOpen      ;get a device handle for monitors for
                     ; the application

CALL DosMonReg       ;register a pair of input and output
                     ; buffers as a monitor

WHILE [we want to monitor the data stream]

    CALL DosMonRead  ;read a data record from the monitor's
                     ;input buffer

    ;process, or filter, the data record (keystroke)

    CALL DosMonWrite ;write a filtered data record into the
                     ;monitor's output buffer

END WHILE

CALL DosMonClose     ;close all monitors from this
                     ;application registered on the same
                     ;device handle
```

Below is a diagram that shows how monitors fit into the system structure and how OS/2 supports these functions.

There are three interfaces in this diagram. They are labeled:

1. Program to device driver interface
2. Device driver to monitor dispatcher interface
3. Monitor dispatcher to monitor process interface



Interfaces

1. Program to device driver

   This interface establishes a monitor connection and tears it down. This is the registration process.

   - Open: DosMonOpen gets a handle to use for monitor registration.
   - Registration: DosMonReg registers monitor and data buffers.
   - Deregistration/Close: DosMonClose deregisters the monitor.

2. Device driver to monitor dispatcher

   This interface is provided with DevHlp routines to do the following:

   - MonitorCreate: Creates a chain of monitors.
   - Register: Adds a monitor to a chain.
   - Deregister: Removes a monitor from a chain.
   - MonWrite: Passes data records to a monitor.
   - MonFlush: Removes data from the monitor chain.

3. Monitor dispatcher to monitor process

   The monitor dispatcher coordinates movement of data between the device driver and monitor input/output buffers.

Module Description

- (A) Monitor Dispatcher: This package is common to all device drivers and serves the device drivers and the monitors in user space.
- (B) Monitor Buffer Management: Applications are responsible for allocating their own monitor buffers, and setting the first WORD of each buffer equal to the length of the buffer. The monitor dispatcher starts and manages all buffers for registered monitors.

Device Monitor Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The device monitor function calls are summarized as follows:

DosMonOpen	Open a connection to an OS/2 device monitor.
DosMonClose	Close a connection to an OS/2 device monitor.
DosMonReg	Register a set of buffers as a monitor.
DosMonRead	Read data from a monitor structure.
DosMonWrite	Write data to a monitor structure.

Monitor Data Structures

The monitor data structures consist of buffers and data records.

The monitor API and monitor dispatcher device helper routines manage the buffers for monitor applications. Applications registering monitors are required to provide the length (number of bytes) of the buffer in the first WORD of each buffer, length WORD included. Thereafter, the application must not corrupt the buffer at any time.

Since more than one monitor from more than one process may be registered as monitoring the same data stream, a minimum buffer length must be defined to maintain data movement through all buffers. The monitor dispatcher defines this minimum buffer length as the length of the device driver's monitor chain buffer plus 20 bytes. The length of a monitor's input and output buffers must be at least this length. This is also the recommended length of the private data area specified on a DosMonRead call.

In/Out Buffers specified on DosMonReg:

Field	Size
Length	2 Bytes
Used by Monitor Dispatcher	18 Bytes
Must be > = Length of device driver monitor chain buffer plus 20 Bytes	

Device driver monitor chain buffer specified on DevHlp MonitorCreate:

Field	Size
Length	2 Bytes
Max read/write record size	

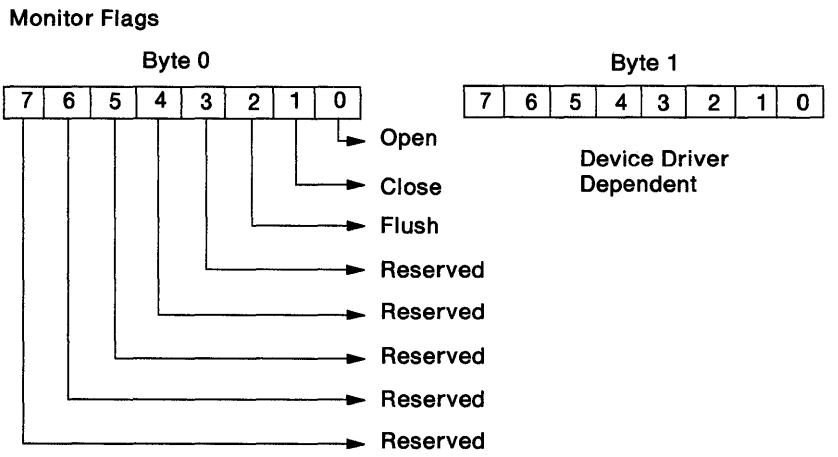
Refer to pages 9-59, 9-93, and 9-155 for device driver specific information.

Device Monitor Record

Only one data record can be read (DosMonRead) or written (DosMonWrite) at a time. Data records in any given input or output buffer may be of different lengths.

Because the minimum buffer size for all buffers in a monitor chain is defined in terms of the device driver's monitor chain buffer, the maximum size of a data record is also defined in terms of this buffer's size. A monitor application cannot write (DosMonWrite) into its output buffer a data record whose length is greater than the length of the device driver monitor chain buffer minus two bytes.

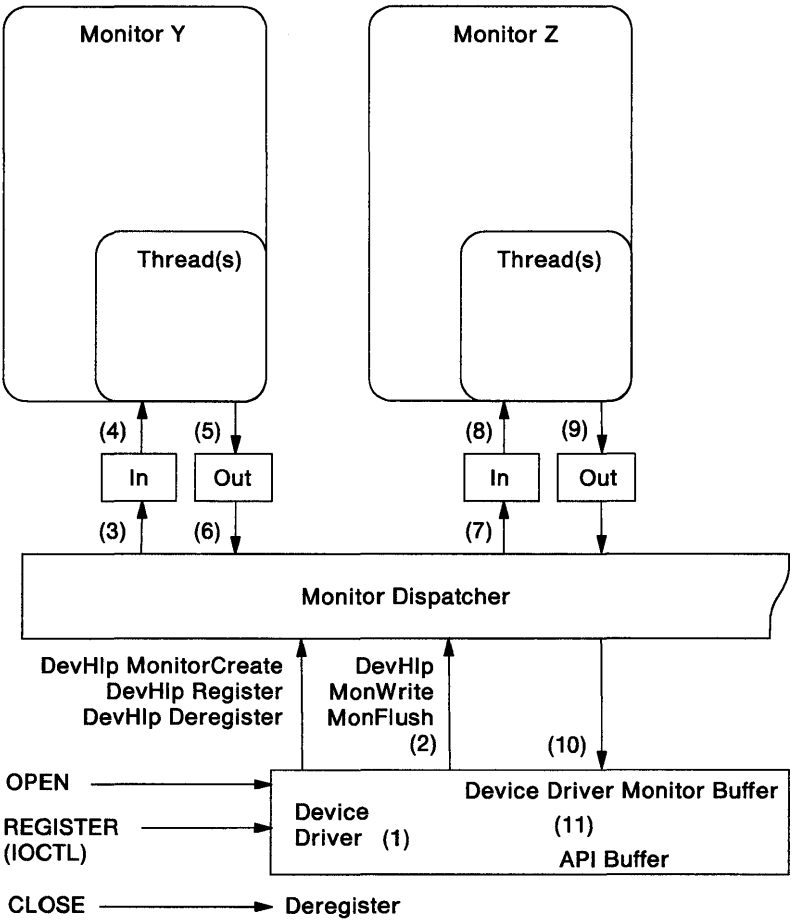
The first WORD of a data record is a flag word. A data record may consist of only a flag word. The flags are defined as follows:



Flushing a data stream is an important operation. All data must be cleaned out of the data stream at certain times. At these times, a flush record will be placed in the data stream by the device driver by a DevHlp_MonWrite and must pass through all monitors in the chain to allow them to reset their internal states. The flush record will force them to flush all internal buffers. Placement of new data records into the chain by way of a DevHlp_MonWrite will be suspended until the flush record reaches the device driver's input buffer. Therefore, flush records must not be consumed by the monitor.

When an application receives a data record with an open or close bit, the action to be taken depends on the issuing device driver. Refer to Chapter 9, "Device Drivers" on page 9-1 for device driver specific information.

Data Flow Through a Monitor



1. The device driver receives the data and decides to which chain, if any, to direct it.
2. A monitor is registered with the chain. The device driver calls DevHlp_MonWrite (monitor dispatcher device helper routine) to place the data into the chain's first buffer, allocated by the monitor dispatcher when the monitor chain is created.

3. The monitor dispatcher automatically moves the data record from the chain's first buffer into the input buffer of the first monitor in the chain (Monitor Y).
4. Registered Monitor Y calls DosMonRead to get a data record from the data stream and place it into its private data area.
5. After filtering the data record, registered Monitor Y returns the it to the data stream by calling DosMonWrite to write the data record into its output buffer.
6. The monitor dispatcher automatically moves the data record from the output buffer of Monitor Y into a monitor dispatcher data area.
7. The monitor dispatcher automatically moves the data record from the monitor dispatcher data area into the input buffer of the next monitor in the chain, Monitor Z.
8. Registered Monitor Z calls DosMonRead to get a data record from the data stream and place it into its private data area.
9. After filtering the data record, registered Monitor Z returns it to the data stream by calling DosMonWrite to write the data record into its output buffer.
10. The monitor dispatcher automatically moves the data record from the output buffer of Monitor Z into the device driver's monitor chain buffer.
11. The device driver moves the data from the device driver's monitor chain buffer into its API buffer, where it is ready to be read by an application.

The Time Window of Monitor Registration

An application that requires monitoring all characters passing through a device, including those received by the device from the time the application is invoked through the time the monitor is registered and has gained access to the data stream, must do some additional work. (For example, the case of type-ahead characters.) The diagram on page 6-52 illustrates the time window between invocation and completion of monitor registration when characters received by the device are placed into the device's API buffer until monitor registration is completed. Monitor registration is completed when characters received by the device are then placed into the monitor chain. The following pseudo code suggests a possible solution for such an application:

```
;OUR APPLICATION WISHES TO MONITOR ALL CHARACTERS TO A
;  DEVICE, INCLUDING TYPE-AHEAD CHARACTERS

;   APPLICATION IS RUNNING
; If no other monitors registered for this chain, data
; to device is being written to device's API buffer

CALL DosMonOpen          ;get handle to the device

;We want to monitor all data received by the device since
; the application was invoked, including type-ahead data

CALL DosMonReg           ;register a monitor with the data stream
   ;If DosMonReg returns no error, then data to device
   ; is being written to monitor chain (i.e. the input
   ; buffer of the first monitor in the chain)

IF [DosMonReg returned without error]
THEN

  ;The device driver will start placing data received from
  ; the device into the monitor chain.  The monitor
  ; dispatcher will start moving data into the input buffer
  ; of the first monitor in the chain.
```

```
  ;Before the DosMonRead/DosMonWrite thread is started, we
  ; want to look at any type-ahead data that was received
  ; by the device BEFORE completion of monitor registration
  ; and make it available to another application.
  ;First, get any type-ahead data

WHILE [device's API buffer is not empty]

   Get a character from the API buffer through the
      Device Subsystem Call

   Place it in a temporary buffer
END WHILE

;Now, make it available to another application by placing it
; back in the device's data stream through our monitor's
; output buffer.

WHILE [temporary buffer is not empty]

   Get a character from the temporary buffer

   Build a device-specific monitor record

   CALL DosMonWrite     ;return it to the device's data
                        ; stream and, therefore, to the
                        ; API buffer for another app
END WHILE

;Begin monitoring data stream - we may want to do this on
; a separate thread

WHILE [we want to monitor data stream]

   CALL DosMonRead      ;get a data record from data stream

   ;process the data

   CALL DosMonWrite     ;return the data to the data stream
END WHILE

CALL DosMonClose        ;close all monitors from this process
                        ; registered on the same device handle
ENDIF
```

Hints for Using Monitors

Some common problems occur in OS/2 monitor applications because of failure to follow the guidelines outlined earlier.

1. The monitor input and output buffers specified when you call DosMonReg should be defined separately, but within the same segment. These buffers should be at least as large as the device driver's monitor chain buffer plus 20 bytes. The first WORD of each buffer should include the length of the buffer in bytes, length WORD inclusive. If the buffers are incorrectly specified, DosMonReg will return an error.

2. A monitor process should be written so the thread(s) that actually read and write the monitor data run at the lowest time-critical priority. They should never perform operations such as I/O or semaphore waits that might delay them. The monitor process can have other thread(s) running at normal priorities to handle such things.

3. Complex processing should be done in threads separate from your monitor (DosMonRead/DosMonWrite) threads. If not, the processing may block the device's data stream. For example, an application running in the same session may never receive data on a read if complex processing is not done separately.

4. The monitor should not consume flush records but, should return them to the data stream by calling DosMonWrite. If the flush records are not returned, the monitor is blocking the data stream and the device driver may send a "buffer full" signal. For example, in the case of the keyboard, a beep will sound.

5. Threads in your application should not poll the device's API buffer to ensure it is never time slicing or yielding so that other threads may run. If polling is done in this fashion, threads will not run, data will not reach the device driver's API buffer, and applications will not find any data available when doing a read. This may cause the system to stop running.

6. A single process can register more than monitors. In addition, the process can distribute these monitors on different chains.

Providing Monitor Support in a Character Device Driver

A character device driver may define more than one data stream for its device. Each data stream may be monitored by a "chain" of one or more registered device monitors. Some devices drivers, such as the keyboard and the mouse, may define their data streams on a per session basis. Other device drivers, such as the printer, may define their data streams on a per device basis.

When an OS/2 character device driver receives data from its device, the driver decides where to direct it, that is, into which of its data streams the driver will place the data. For example, when the keyboard device driver receives a keystroke from the keyboard, it directs that keystroke into the data stream for the current active session.

A character device monitor is an application or part of an application that uses standard OS/2 function calls to interact with the device driver to gain access to one of its data streams for monitoring or filtering all data passing through the device.

In order for an application to monitor data passing through a character device, the character device driver must provide monitor support.

A device driver must include in its *DATA segment*:

1. A buffer in which to build the data record placed into the monitor chain on a DevHlp MonWrite call.

   DS:SI (on calling DevHlp MonWrite) is the address of the data record to be placed into the monitor chain. DS must point to the device driver's data segment when DevHlp MonWrite is called.

2. For each data stream that can be monitored by a chain of monitors, a "monitor chain buffer" that will receive filtered data from the chain of monitors; that is, from the output buffer of the last monitor in the chain.

   The first WORD of each monitor chain buffer must first (for example, before calling DevHlp MonitorCreate) contain the LENGTH of the buffer (length word inclusive). The length of a monitor chain buffer will define:

   a. The minimum size of all buffers that are part of the monitor chain; that is, the length of each of the input and output buffers of all monitors that register with the chain must be greater than or equal to the length of the device driver monitor chain buffer plus 20 bytes.

   b. The maximum size of a data record placed into the chain of monitor buffers (that is, the length of the device driver buffer minus 2 bytes). See diagram on page 6-50.

A device driver must include in its *CODE segment*:

1. A *NOTIFICATION routine* that is called by the monitor dispatcher when the monitor dispatcher has placed a single 'filtered' data record that has passed through the entire monitor chain into the monitor chain buffer. The device driver must process this data record before returning to the monitor dispatcher; for example, make it available to device subsystem calls by moving it into the API buffer.

   When the *NOTIFICATION* routine has been called by the monitor dispatcher,

   a. the first WORD of the monitor chain buffer contains the length (length word inclusive) of the filtered data record. See diagram on page 6-50.

   b. DS points to the device driver's data segment and ES:SI points to the device driver's monitor chain buffer.

For each data stream that can be monitored by a chain of monitors a device driver must define for the monitor dispatcher the location (addresses) of the notification routine and the monitor chain buffer by calling DevHlp MonitorCreate:

1. The monitor dispatcher assigns a handle to the chain of monitors for the data stream. The device driver will use this handle when instructing the monitor dispatcher to perform device helper functions on the chain of monitors (that is, DevHlp Register, DevHlp MonWrite, DevHlp MonFlush, and DevHlp Deregister).
2. The device driver may call DevHlp MonitorCreate any time prior to issuing other monitor dispatcher device helper functions:
   a. When the device is loaded;
   b. In response to an 'open' request, if DevHlp MonitorCreate has not been previously called to define the monitor chain; or
   c. In response to a 'register' request, if DevHlp MonitorCreate has not been previously called to define the monitor chain, and before DevHlp Register is called.

A device driver must handle open, register, and close requests for monitors.

1. When an application calls DosMonOpen to get a handle to the device for monitors, an open request is sent to the device driver. In response, a device driver may define a monitor chain for the data stream by calling DevHlp MonitorCreate, if it has not previously done so.
2. When an application calls DosMonReg to register a pair of buffers as part of the chain of monitors for a specified data stream (see DosMonReg INDEX parameter in *Technical Reference, Vol. 2*) for a device, a register request is sent to the device driver. In response, the device driver:
   a. May call DevHlp MonitorCreate to define the monitor chain, if it has not previously done.
   b. Uses the handle returned from a previous DevHlp MonitorCreate call and calls DevHlp Register, to direct the monitor dispatcher to insert the buffers into the chain of monitors for the specified data stream.

      For each monitor chain handle, the device driver should track the number of monitors it registered with the monitor chain.

3. When an application calls DosMonClose to terminate monitoring data passing through a device, a 'close' request is sent to the device driver. In response, the device driver:

   a. Must call DevHlp Deregister for each monitor chain handle for monitors currently registered. This directs the monitor dispatcher to remove all buffers associated with the process that issued the DosMonClose from the chain of monitors for the data stream.

      Note: A single process may register monitors for more than one data stream for a device. For example, a process may register a keystroke monitor for each session.

   b. May call for a monitor chain on return from each DevHlp Deregister. If there are no monitors registered in the monitor chain, call DevHlp MonitorCreate with the "remove chain" option to remove the monitor chain's definition from the monitor dispatcher's knowledge.

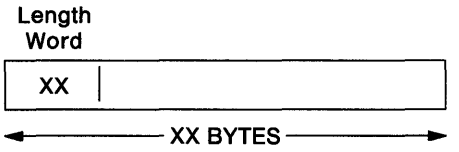
When monitors are registered in a chain of monitors for a device driver's data stream, the device driver must place data received from the device into the chain of monitors so that it can be "filtered." The device driver builds a data record in its data segment and calls DevHlp MonWrite to write that record into the input buffer of the first monitor in the monitor chain for the data stream. See page 6-52 for a detailed description.

When a chain of monitors for a device driver's data stream is empty (that is, a chain has been defined via a DevHlp MonitorCreate call, but no monitors have been previously registered), the device driver may use its monitor support to place a data record directly into its monitor chain buffer by calling DevHlp MonWrite. The monitor dispatcher automatically will call the device driver's notification routine to move the data record into the device driver's API buffer. See page 6-51 for a detailed description.

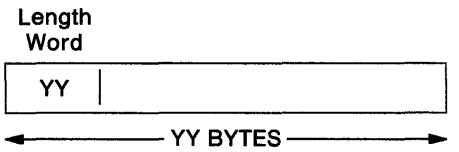
When a device driver needs to guarantee that all data has been cleaned out of the chain of monitors for a data stream, it issues a DevHlp MonFlush call to direct the monitor dispatcher to place a specially marked record, a FLUSH record, into the chain of monitors for the data stream. This record must pass through all monitors in the chain; therefore, all monitors in the chain that receive a FLUSH record on a DosMonRead must return it to the monitor chain on a DosMonWrite. No new data records will be placed into the monitor

chain until the FLUSH record reaches the device driver's monitor chain buffer.

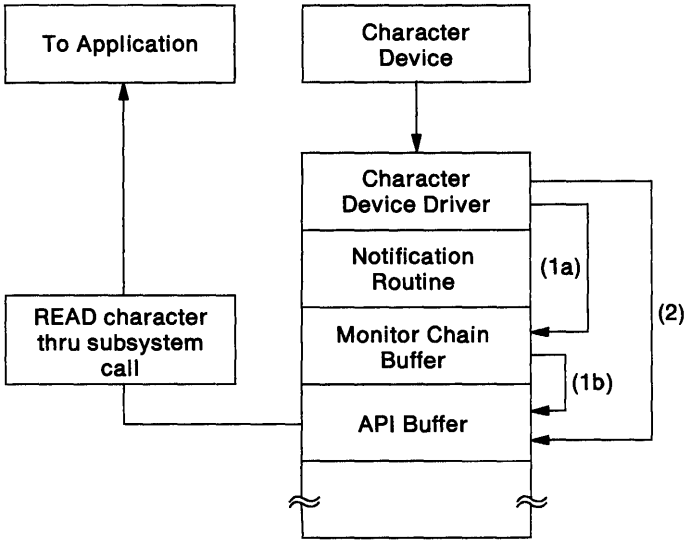
The following diagram shows a device driver monitor chain buffer before a data record is placed into it by the monitor dispatcher.



The following diagram shows a device driver monitor chain buffer after a data record is placed into it by the monitor dispatcher. The YY length is less than or equal to the length of the device driver monitor chain buffer as initially defined.



The following diagram shows data movement through a character device driver before monitor registration:

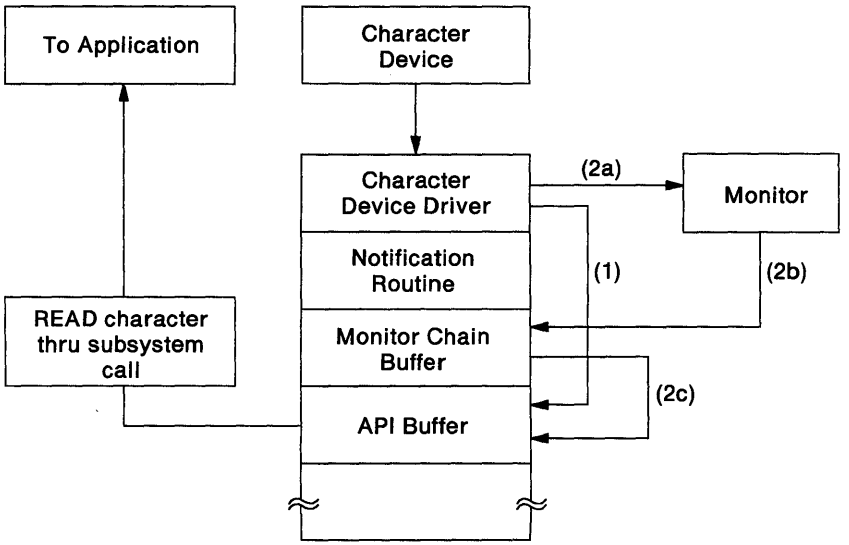


Note: No monitors are installed.

Data received by the device by an INT or a write passes through the device driver to its API buffer. The device driver has an option to do this by either:

1. Using the existing device driver monitor support to:
   a. Write the data into the Monitor Chain buffer using DevHlp MonWrite and let the monitor dispatcher automatically call the device driver's notification routine (to signal the device driver that data has been written into this buffer);
   b. Let the device driver's notification routine move the data into its API buffer.
2. Writing the data directly into its API buffer.

The following diagram shows data movement through a character device driver during and after monitor registration:



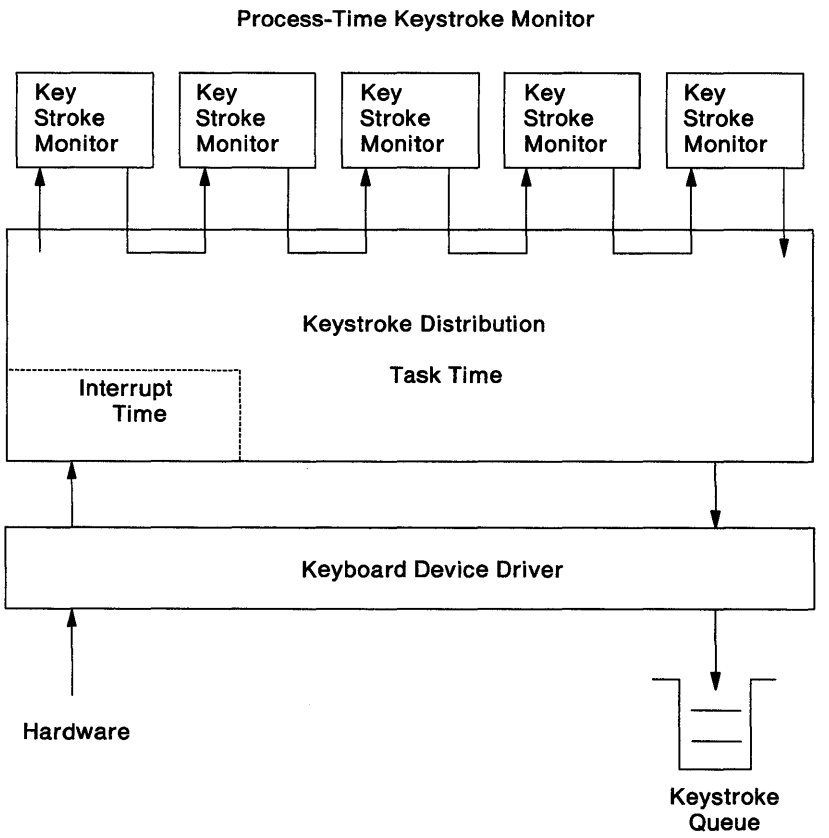
1. A process issues a DosMonReg call to register a monitor with a character device. Until the monitor registration process has completed successfully, data is written into the device driver's API buffer.
2. Once the monitor registration process has completed successfully,
   a. Data is placed into the monitor chain by the device driver using DevHlp MonWrite.
   b. After the data has passed thru the monitor using DosMonRead/DosMonWrite calls, the monitor dispatcher moves the data into the device driver's monitor chain buffer and calls the device driver's notification routine to signal it that data has been placed into this buffer.
   c. The device driver moves the data from its monitor chain buffer into its API buffer.

Keystroke Monitor Interface

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

This diagram depicts the keyboard device keystroke monitor interface. The keystroke monitors have been previously enrolled as monitors. A monitor can pass the keystroke through, consume the keystroke, or replace the keystroke with one or many keystrokes.

Keystroke Monitor Interface Diagram

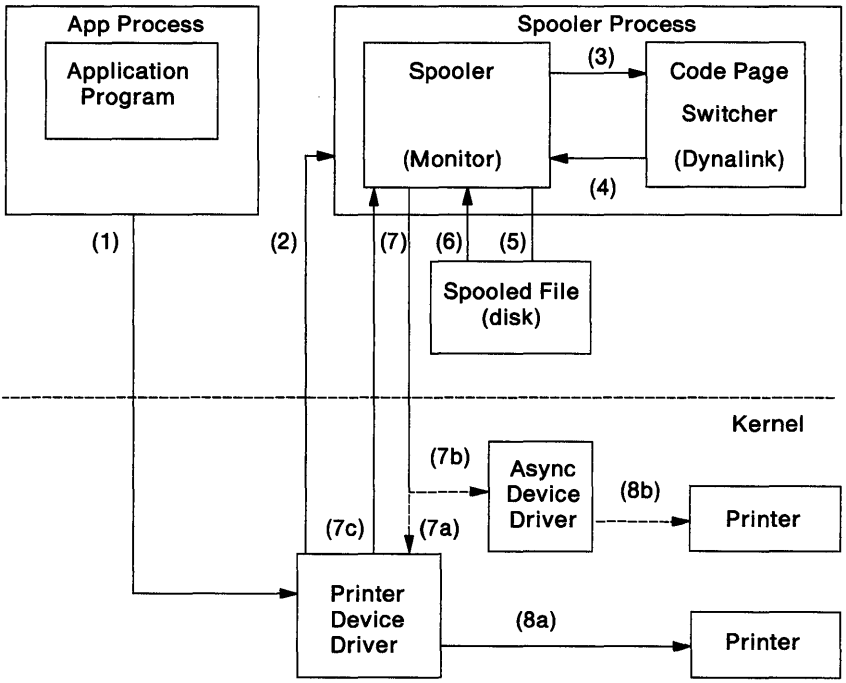


Threads responsible for moving keystroke data through a monitor chain must pay special attention to the thread priority. Keystroke monitor threads must execute within the time critical priority class. See *Technical Reference, Vol. 2* description of DosMonReg for more details concerning keystroke monitor priority.

Printer/Spooler Services

The printer/spooler is structured as shown in the diagram below:

Printer/Spooler Structure



The data flow shown in the printer/spooler structure above is:

1. This flow is the normal API to the printer.

   The Application program issues DosOpen, followed by "n" DosWrites/DosIOCtls to the printer. The file system sends an Open request packet to the printer device driver, followed by an Activate Font IOCtl. The Application program's DosWrites go to the printer device driver until a DosClose is issued.

2. This flow is from the printer device driver to the spooler through the DosMonRead monitor interface.

The printer device driver sends an Open command buffer, followed by an Activate Font monitor command buffer, and then "n" buffers corresponding to the DosWrites.

3. This flow is by a Call interface to the Font Switcher dynamic link routine.
    a. This occurs when the Activate Font, Query Active Font, or Verify Font command buffer is received by the spooler from the printer device driver.
    b. Escape sequence data and/or font data necessary to cause the printer to perform the actual code page and font switch is written to the temporary spool file by the Font Switcher. The data is treated by the spooler as part of, and in sequence with, the data being printed by the application program.
4. This flow is the return from the call described in the flow previously.
5. This flow shows data being written by the spooler to a temporary spool file using DosWrites. This is the data sent by the application program to be printed.
6. The spooler reads data from its temporary spool files on the disk, in sequence based on its spool queues, using DosReads.
7. The spooler sends data to the device driver to be printed.
    a. If the spooled printer is a parallel device, the spooler sends the data back to the printer device driver in a monitor buffer using Monitor Index 1 for the DosMonWrite.
    b. If the spooled printer is an ASYNC device, the spooler sends the data back to the ASYNC device driver using the DosWrite function interface.
    c. If the data to be returned is a Font Monitor Buffer Response, the spooler sends the response back to the printer device in a monitor buffer using Monitor Index 2 for DosMonWrite.
8. The device driver sends data to the printer through the hardware adapter.
    a. The printer device driver sends the data to parallel printers.
    b. The ASYNC device driver sends the data to serial printers.

Spooler Monitor

DOS Mode Force Output to Printer

The print spooler spools the printer output until a DosClose is issued. When printing from the DOS mode, many applications use the INT17 interface which does not require DosOpen/DosClose. In this case, the printer output is spooled until the program exits. The user may press Ctrl-Alt-PrtSc (while the DOS mode is in the foreground) to force printed output to be printed from DOS mode applications.

Spooler Operational Description

The Spooler uses a dynamic link module called the Font Switcher to perform the code page and font switching.

Initialization: When the Spooler is invoked by the user (user command SPOOL), it accesses the system data structure which contains the code page and font information provided by the DEVINFO command in CONFIG.SYS. If DEVINFO is present for the specified printer name (LPTx) in the code page and font data structure, the spooler will call the DosPFSInit entry point of the Font Switcher dynamic link module to initialize font switching. After the Spooler and Font Switcher have completed initialization, print spooling along with code page and font switching are enabled for the specified printer.

If the DEVINFO code page and font information are not present in the code page and font data structure maintained by the system for the specified printer name, the spooler completes its initialization without initializing the Font Switcher for the specified printer. In this case spooling is enabled for the specified printer, however, code page and font switching are not.

Activate Font: When an Activate Font - Font Monitor Buffer Command is received by the spooler for a specific System File Number, the spooler calls the DosPFSActivate entry point of the Font Switcher. The Font Switcher changes the active code page and font for the System File Number(/handle), and uses the font file to cause the code page and font switch to occur. The code page and font switch occurs in one of the following ways depending on the font support provided by the target printer:

1. If the specified font is contained in the printer hardware (ROM or cartridge, as specified by the DEVINFO command CONFIG.SYS), the Font Switcher writes the escape sequence required to switch the printer to this hardware font directly into the temporary spool file. The escape sequence necessary to perform this switch is available for the Font Switcher in the font file specified for the printer.

2. If the specified font is not contained in the printer hardware, and the printer allows multiple downloadable fonts, the Font Switcher will write the following information directly into the temporary spool file in the following order:

   a. The escape sequence required to cause the font data that follows to be loaded into the correct printer buffer;

   b. The font information;

   c. The escape sequence required to cause the printer to use the buffer just loaded.

   If the Activate Font command specifies a font which has been previously loaded into a font buffer other than the one which is currently active, the Font Switcher writes the escape sequence required to switch the printer to the desired font buffer directly into the temporary spool file.

   All escape sequences and font information required are available for the Font Switcher in the font file specified for the printer.

3. If the specified font is not contained in printer hardware, and the printer allows a single font to be downloaded, the Font Switcher will write the following information directly into the temporary spool file in the following order:

   a. The escape sequence required to cause the font data that follows to be loaded into the correct printer buffer;

   b. The font information;

   c. The escape sequence required to cause the printer to use the buffer.

   All escape sequences and font information required are available for the Font Switcher in the font file specified for the printer.

Query Active Font: When a Query Active Font - Font Monitor Buffer Command is received by the spooler for a specific System File Number, the spooler calls the DosPFSQueryAct entry point of the Font Switcher. The DosPFSQueryAct function returns the active code page and font for the specified System File Number(/handle) to the spooler. The spooler returns the information to the printer device driver using the Query Active Font Font Monitor Buffer Response. The printer device driver then returns the information to the Query Active Font IOCtl caller.

Verify Font: When a Verify Font - Font Monitor Buffer Command is received by the spooler for a specific System File Number, the spooler calls the DosPFSVerifyFont entry point of the Font Switcher. The DosPFSVerifyFont function returns whether the specified code page and font is available for the specified printer.

Spooler Monitor Interfaces

The interfaces required by the spooler for code page and font switching are:

- System code page and font information
- Font Monitor Buffer Commands
  - Activate Font
  - Query Active Font
  - Verify Font
- Font Switcher function calls are summarized as follows:

DosPFSInit	Initialize code page and font
DosPFSActivate	Activate Font
DosPFSQueryAct	Query Active Font
DosPFSVerifyFont	Verify Font
DosPFSCloseUser	Close Font User Instance

  For function call details, refer to *Technical Reference, Vol. 2.*

- Printer Font File Format

These interfaces are described in the following sections.

System Code Page Information: The spooler must be able to access the System Code Page Information when it is invoked (user command SPOOL) from some system provided data segment. The information required is that provided by the CONFIG.SYS command DEVINFO.

This information includes the printer name, printer type, code page and fonts provided in the printer hardware and the path name of the font file to be used.

Font Monitor Buffer Commands: The Font Monitor Buffer Commands which provide the code page and font switch interface between the printer device driver and the spooler monitor are:

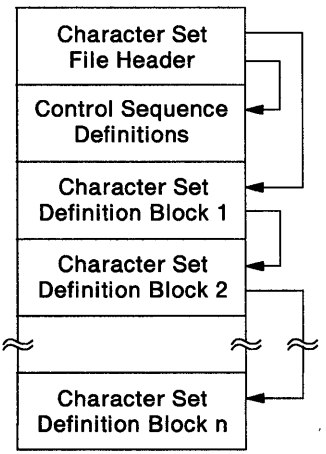
1. Activate Font
2. Query Active Font
3. Verify Font

These commands and their responses are described in detail in "Font Monitor Buffer Commands" on page 9-159.

Printer Font File Format: Each printer for which code page and font switching is supported must have a font file which describes:

- Control Sequences for downloading and switching fonts
- Font definitions

OS/2 Font File Format
The format of the OS/2 Font File is shown in the following diagram:



The Font File Header defines the printer which the font file supports and has pointers to the Control Sequence Definitions section of the file and to the first Font Definition Block. The Font Definition Blocks

are a linked list throughout the rest of the file. Each of these sections of the font file are described in more detail below:

Font File Header

The format of the font file header is shown in the following diagram:

00	Font File Type Number
02	Printer Type
0A	Version Major Number
0B	Version Minor Number
0C	Number of Hardware Fonts
0D	Number of RAM Fonts
0E	Control Sequence Blk Ptr
10	Character Set Def Blk Ptr
14	Reserved (46 Bytes)

The definition of each field is as follows:

Font File Type Number This field identifies the format version or type of the font file. The value for the format specified for OS/2 is **4554h.**

Printer Type This field is eight bytes which specify the printer type which this font file supports.

Version Major Number This byte is the "major" version number of the font file.

Version Minor Number This byte is the "minor" version number of the font file.

Number of Hardware Fonts This byte specifies the maximum number of hardware fonts (ROM or cartridge slots) which this printer supports.

Number of RAM Fonts This byte specifies the maximum number of fonts which this printer may have downloaded to it simultaneously.

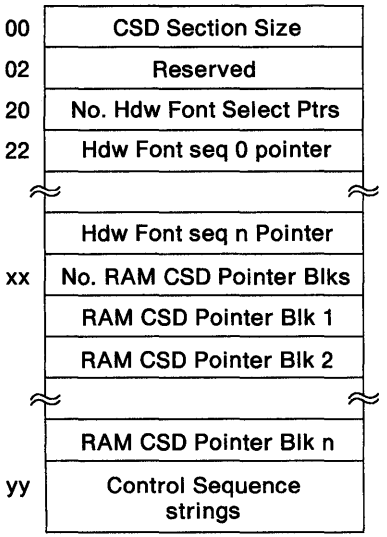
Control Sequence Blk Ptr This WORD pointer is the offset from the beginning of the file at which the Control Sequence Definition Section begins.

Character Set Def Blk Ptr This DWORD pointer is the offset from the beginning of the file at which the first Font Definition Block begins.

If this value is zero (0), there are no font definition blocks in this file.

Reserved This area of 46 bytes is reserved and should be set to zero (0).

Control Sequence Definitions: The following diagram shows the format of the font file control sequence definition section:



The definition of each field is as follows:

CSD Section Size Control Section Definition Section Size - This WORD gives the total size in bytes of the CSD Section (including itself)

Reserved This area is reserved and must be set to zero.

No. Hdw Font Select Ptrs This Word specifies the number of Hardware Font Select Sequence DWORD pointers which follow.

Hdw Font Select x ptr Each DWORD pointer in the Hardware Font Select Sequence Table points to the beginning of a length prefixed control sequence string which is in the "Control Sequence Strings" area of this section.

The first control sequence corresponds to the control sequence command for "Select Hardware Font 0", the second control sequence is for "Select Hardware Font 1", and so on. Hardware Font 0 corresponds to the hardware default font of the printer.

For printer type 5202 only, the control sequences do not correspond to hardware fonts except for the first control sequence which is necessary to select the hardware font when both code page and font id are zero. If the printer type is 5202, the font file is for a printer with plug-in cartridge fonts and the hardware font select sequences are to select a cartridge code page, font id, pitch, and whether proportional spacing is used. These items are dynamically inserted into the control string at runtime.

When an Activate Font command is specified with both the OS/2 values set to zero, then Hardware Font 0 is selected.

Each pointer is an absolute DWORD offset from the beginning of the file.

No. RAM CSD Pointer Blks This Word specifies the number of RAM Font Buffer Control Sequence Definition Pointer Blocks (RAM CSD Pointer Blk) which follow.

RAM CSD Pointer Blk x The RAM Font Buffer CSD Pointer Blocks are an array in which each element (block) contains the control sequences which correspond to a font buffer in the printer. The order of the blocks correspond to the RAM font buffer number (that is the first block corresponds to the first font buffer, and so on). Each pointer in a RAM CSD Pointer Block is an absolute DWORD offset from the beginning of the file to length prefixed control sequence strings contained in the "Control Sequence Strings" area of this section. Any pointer which is zero (0) indicates that the corresponding control sequence is not defined.

RAM Font Buffer CSD Pointer Block Format

Each RAM CSD Pointer Block has the following format:

00	Select Buffer Seq Ptr
04	Begin Download Seq Ptr
08	End Download Seq Ptr
0C	Reserved (20) Bytes)

Select Buffer Seq Ptr The Select Buffer Sequence DWORD pointer points to the beginning of a length prefixed control sequence string which is in the "Control Sequence Strings" area of this section. This control sequence is used to "select" (or make active) the font buffer. This pointer is an absolute offset from the beginning of the file.

Begin Download Seq Ptr The Begin Font Download Sequence DWORD pointer points to the beginning of a length prefixed control sequence string which is in the "Control Sequence Strings" area of this section. This control sequence specifies that a font download for a particular font buffer follows. This pointer is an absolute offset from the beginning of the file.

End Download Seq Ptr The End Font Download Sequence DWORD pointer points to the beginning of a length prefixed control sequence string which is in the "Control Sequence Strings" area of this section. This control sequence specifies that a font download for a particular font buffer is complete. This pointer is an absolute offset from the beginning of the file.

Reserved Each RAM Font Buffer CSD Pointer block contains a reserved area of 20 bytes which must be set to zero (0).

Control Sequence Strings This area of the section contains the actual length prefixed control sequence strings pointed to by the pointers described above. The first byte of each string indicates the length of the string in bytes.

If the printer type is 5202, there are dynamic inserts for cartridge code page, font id, pitch, and proportional spacing. These inserts are made at the following positions of the control string (starting counting with one):

Byte	***Description***
9	High order byte of code page
10	Low order byte of code page
16	High order byte of font ID
17	Low order byte of font ID
18	High order byte of pitch (1/1440 inches)
19	Low order byte of pitch (1/1440 inches)
20	Proportional spacing on/off

Value	***Description***
0 =	Wild card
1 =	Use fixed pitch
2 =	Proportional

Font Definition Block

Each font definition block within the file has the following format:

00	Pointer to Next Block
04	Code Page Number
06	Font Id Number
08	Length of Font Data
0A	Reserved
20	Font Definition Data

The definition of each field is as follows:

Pointer to Next Block This DWORD pointer points to the next font definition block in the file.

A value of zero (0) indicates that this is the last font definition block in the file. This pointer is an absolute offset from the beginning of the file.

Code Page Number This is the Code Page number of the font definition contained within this block.

Font Id Number This is the Font Id number of the font definition contained within this block.

Note: The code page number and font id should not both be zero because this indicates the default hardware OS/2. If both are zero, the font will not be selected.

Length of Font Data This is the number of bytes in the "Font Definition Data" area.

Reserved These bytes are reserved, and must be set to zero (0).

Font Definition Data This area contains the bytes which are sent to the printer when downloading a particular font.

Chapter 7. OS/2 Device Driver Architecture

Device drivers written for DOS are synchronous and often non-interrupt driven. Because DOS is a single-task operating system, this presents no problem. A program cannot proceed until the I/O has been completed, so it is acceptable for the device driver to hold the CPU until the I/O is complete.

OS/2, on the other hand, is a multitasking operating system. It must be able to assign the CPU to tasks that are not waiting for I/O. Therefore, device drivers written for OS/2 must surrender the CPU while they are waiting for I/O completion. Consequently, OS/2 device drivers must be interrupt-driven.

In addition, OS/2 provides a DOS mode to support DOS applications. Consequently, OS/2 device drivers must support device I/O requests in the DOS mode. This means that OS/2 device drivers must handle I/O requests issued by the applications running in the DOS mode and may need to interlock DOS mode ROM BIOS device I/O with OS/2 mode device I/O. Performance considerations also require that device drivers be able to handle hardware interrupts without the overhead of a mode changing method. OS/2 device drivers must therefore be bimodal, that is, execute in both the OS/2 mode and the DOS mode.

The basic characteristics of the OS/2 device driver are:

- Support of a multitasking environment
- Execution in both the DOS mode and the OS/2 mode

Types of Device Drivers

There are two types of device drivers:

- Character device drivers
- Block device drivers

Character device drivers manage I/O on character-oriented devices. These devices perform I/O on a character-by-character basis. A character device has a name like SCREEN$, KBD$, LPT1, or COM1. A character device driver can support more than one device by having multiple linked device headers where each header indicates a different name.

Block device drivers support block-oriented devices. These devices perform I/O on a block of data, typically through DMA (Direct Memory Access). Applications request I/O on block-oriented devices via a file system. Consequently, block device drivers do not have names. Instead, a block device driver is assigned one or more drive letters. A block device driver can support multiple devices (or units), so a drive letter is assigned for each unit (or device). A block device driver specifies the number of devices that it supports when it is called to initialize. Refer to the INIT device command, "0H / INIT Initialize Device" on page 7-43.

The order in which the block device drivers appear in the CONFIG.SYS configuration file (via the DEVICE= command) determines the order in which they receive drive letters.

Application I/O to Devices

An application accesses a device driver when it performs an I/O request. There are three kinds of interfaces that provide access to device drivers:

- File system interface
- Subsystem interface
- IOCtl interface

The File system interface consists of the file I/O function calls. OS/2 applications use the DOS dynamic link calls for file I/O, DOS applications use the DOS dynamic link calls for file I/O, and DOS applications running in the DOS mode issue INT 21H File I/O function calls. The file I/O function calls are used primarily to perform I/O on block devices like fixed disks. Some file I/O function calls can be used to perform I/O on character devices. These calls include:

- DosOpen
- DosClose
- DosRead
- DosReadAsync
- DosWrite
- DosWriteAsync

The advantage of using the file I/O function calls to perform I/O on a character device is that the application can redirect the I/O. The application performs I/O to a handle (file or device) which it obtained from opening the named resource passed to it.

A subsystem interface shields an application from having to manage device I/O. The file system is an example of a subsystem. The file system allows the application to view a file as a logical sequence of sectors, thus shielding the application from having to manage the physical locations of the data on the disk media. Other subsystem examples include the VIO, KBD, and MOU subsystems which provide I/O services for video, keyboard, and mouse devices respectively.

The IOCtl interface is the mechanism that an application or subsystem uses to send device-specific control commands to the device driver. The IOCtl mechanism is DosDevIOCtl for new applications

and the INT 21H IOCtl request for DOS applications. I/O commands can be sent to both block and character device drivers. The application or subsystem must first obtain the device handle by doing an open on the device name.

I/O Support For The DOS Mode

A device driver is responsible for managing I/O for its device. Because many devices must be accessible from both the DOS mode and the OS/2 mode, an OS/2 device driver must manage its device across both modes of operation. Examples of such cross-mode devices are disk, keyboard, screen, mouse and printer.

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

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

I/O using the DOS INT 21H interface is transformed into the request packet interface, which the device driver receives. (Similarly, I/O using the OS/2 dynamic link function calls are transformed into the request packet interface.) Because the request packet interface is standard across modes for new device drivers, it poses no problem for the device driver in managing its device.

I/O using the ROM BIOS poses some problems for an OS/2 device driver. The OS/2 device driver must intercept the ROM BIOS software interrupt (by setting the vector with the DevHlp SetROMVector — see "SetROMVector Set DOS Mode Software Interrupt Vector" on page 8-78) and interlock ROM BIOS operations on its device in two ways:

- Serialize access to the device.

  Serialize 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 suspension.

  Suspension occurs when a user switches away from the application in the DOS mode to an application in the OS/2 mode. This causes the DOS mode to be suspended in the background. However, some I/O processing 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 OS/2 device driver to intercept the appropriate ROM BIOS interrupt and issue the DevHlp function call, ROMCritSection, to protect the ROM BIOS critical section of execution.

Note: When the OS/2 device driver issues ROMCritSection to enter a ROM BIOS critical section, the user is not able to switch away from the application in the DOS mode to an application in the OS/2 mode. This poses potential problems for the user. For example, if a DOS 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. The worst case is that the terminate-and-stay resident application is interactive, never allowing the OS/2 device driver to issue the exit from critical section and never allowing the user to switch away from the application in the DOS mode until the user terminates the application.

Application I/O using direct access to a device driver's device poses the same problem for the device driver under OS/2 as it does to a device driver operating under DOS. If device state information is critical or if device I/O must be serialized, then a device driver can 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.

Components of a Device Driver

An OS/2 device driver contains one or more of the following components:

- Strategy Routine

  The strategy routine is called to handle I/O requests through a request packet interface with the OS/2 kernel. The strategy routine executes at task-time as a result of an application I/O request. Because application I/O requests can come from new OS/2 applications running in the OS/2 mode and DOS applications running in the DOS mode, the strategy routine must not have a dependency on the mode in which it is invoked.

  The strategy routine follows the FAR CALL/RET model. When it is done processing, it performs a FAR RET to the kernel. By con-

vention, the strategy routine does not need to save and restore any registers it uses, because the preservation of registers is handled by the kernel.

The request packet interface is discussed in the section "Request Packets" on page 7-37.

- Hardware Interrupt Handler

  The hardware interrupt handler is called as the result of a hardware interrupt and executes at interrupt time. For performance reasons the hardware interrupt handler must not have a dependency on the mode in which it is invoked.

  The hardware interrupt handler follows the FAR CALL/RET model. When it has completed processing, it performs a FAR RET to the kernel. In addition, it must set or clear the CF (Carry Flag) to indicate whether or not it owns the interrupt. By convention, the hardware interrupt handler does not need to save and restore any registers it uses. This is done by the kernel before calling the device driver.

- Timer Handler

  The timer handler is called as the result of a periodic clock tick and executes at interrupt time. The timer handler manages timeouts and is similar to the INT 1CH user timer feature of ROM BIOS. The timer handler must not have a dependency on the mode in which it is invoked.

  The timer handler follows the FAR CALL/RET model. When it has completed processing, it performs a FAR RET to the kernel. The timer handler must save and restore any registers it uses.

- Software Interrupt Handler

  The software interrupt handler is called directly by a software interrupt. Software interrupts can only be issued in the DOS mode, so the software interrupt handler executes only in the DOS mode. Typically, the software interrupt handler is used to intercept ROM BIOS software interrupts to serialize device access between the protect mode and the real mode; or to prevent the BIOS service from being suspended when the user attempts to switch away from the DOS mode application to an OS/2 mode application.

OS/2 Device Driver Contexts

There are four contexts (modes) in which OS/2 device drivers operate. They are:

- Kernel Mode

  The OS/2 kernel calls the device driver strategy routine for task-time operations. (Task-time is a generic term that refers to executing code as a thread within a process.) The strategy routine will execute as a thread within a process. The strategy routine will not be preempted by a task switch but may be interrupted by incoming hardware interrupts. Kernel mode applies to both the DOS mode and the OS/2 mode.

- Interrupt Mode

  The OS/2 kernel calls the device driver interrupt-time components, the hardware interrupt handler, and the timer handler. (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.) The hardware interrupt handler and the timer handler will not execute code as a thread belonging to a thread specific process; the thread of execution results from a hardware interrupt. Interrupt mode applies to both the DOS mode and the OS/2 mode.

- User Mode

  In user mode the device driver software interrupt handler is called, and applies only to the DOS mode. The software interrupt handler is invoked by a software interrupt. In this mode, the device driver software interrupt handler may be preempted by a task switch.

- INIT Mode

  In INIT mode the device driver strategy routine is called with a request packet containing the INIT command. The initialization code runs in the OS/2 mode at the application privilege level with I/O privilege. A limited set of dynamic link function calls are available for use, as well as a portion of the device helper (DevHlp) function calls. This is discussed in the section "Device Driver Initialization" on page 7-26.

OS/2 Device Driver Operations

To show the interaction between the strategy routine and the hardware interrupt handler in the processing of an I/O request, the following example is presented.

The handling of an I/O request begins with OS/2 calling the strategy routine entry point with a request packet. The strategy routine checks the validity of the I/O request. If the request is valid, the strategy routine may place the request on a work queue for the device, using the DevHlp functions for request queue management.

If the device is currently idle, the strategy routine starts the request at the device. The strategy routine may then wait for the device driver interrupt by suspending its thread of execution with the DevHlp function BLOCK.

When the device interrupt occurs, the hardware interrupt handler checks the request to see if it has been completed. If the request has not been completed, the hardware interrupt handler continues the request at the device. If the request has been completed, the hardware interrupt handler sets the return status in the request packet. The hardware interrupt handler may remove the completed request packet from the work queue and start the next request at the device. If the strategy routine is waiting for the device interrupt, (which is blocked), then the hardware interrupt handler can wake up the strategy routine with the DevHlp RUN.

In this example, the strategy routine queues the requests and only initiates the I/O if the device has been inactive; the hardware interrupt handler starts requests as they reach the head of the work queue. The thread context, in which the device driver determines that a particular request is complete, is not necessarily the same thread context in which the device driver received the request. This is particularly true for the interrupt-time components of the device driver. For example, the address of a user buffer passed to the device driver when the request was issued belongs to a specific LDT, which may not be the current LDT when the request ends. The device driver can accommodate this by storing the buffer address as a 32-bit physical address.

The device driver strategy routine is called by OS/2 with a pointer to the request packet. The pointer to the request packet is bimodal: in

other words, the pointer is valid in both the DOS mode and the OS/2 mode. Any addresses passed in the request packets for read/write requests are passed as 32-bit physical addresses (normalized). Therefore, the device driver does not need to lock or convert the addresses into physical addresses. The device driver only needs to lock addresses that it receives from a source other than OS/2, such as in the case of a process passing an address via a generic IOCtl.

The multitasking environment dictates that the components of the device driver must be capable of handling requests simultaneously. This means that the components of the device must relinquish execution whenever possible. The device driver relinquishes control at task-time by BLOCKING, YIELDING, or referencing a segment which had been swapped out; OS/2 will not preempt a thread in the device driver. However, once the device driver releases its execution, OS/2 can call the device driver with a new request. In other words, once the strategy routine BLOCKS, YIELDS, or references a swapped-out segment, its thread of execution can be called with a new request under the context of a different thread.

While the strategy routine can assume that it will not be preempted by other task-time instances, it must protect itself against its own interrupt-time components. It should disable interrupts when checking if the device is active and when examining the device queue. The interrupt-time components will only be preempted by other higher priority interrupts.

One component of the device driver may be preempted, the software interrupt handler. The software interrupt handler is invoked by a software interrupt in the DOS mode. It can be preempted by background OS/2 mode threads which are scheduled to execute and which may issue I/O requests causing other components of the device driver to be invoked.

Request Packet Queue Management

The strategy routine can either queue a request packet or process it immediately. Typically, only read requests and write requests need to be queued because the device is busy. Other types of requests can usually be handled immediately by the strategy routine.

A block device driver such as the disk device driver, can process queued requests in any order. For instance, the block device driver

can choose to sort the requests to optimize device access time. A character device driver must always handle queued requests in the order it received them; otherwise, mixed output could result.

The request packet queue is really a linked list. The request packet contains a linkage field which allows the packets to be chained together.

The device driver can manage its work queue of request packets with the DevHlp functions for Request Queue Management.

In order to use the Request Queue Management DevHlp services, the device driver must allocate a DWORD variable as a queue header, with one queue header per queue. The DWORD variable must be initialized to zero to indicate an empty linked list or the end of the linked list.

The DevHlp services use the queue header to identify a specific linked list of request packets and will set the header to the first request packet in the list. The linkage field in the request packet is then used to chain the request packet to another request packet.

Because the pointer to the request packet is bimodal (valid in both the DOS mode and the OS/2 mode), the device driver can manipulate the linkage fields in the request packets itself by using its own linked list management.

Memory Management

The device driver must manage addressability to data across task time and interrupt time operations. DevHlp services are provided to allow the device driver to be independent of the CPU mode, whether at task-time or interrupt-time. Addressability is particularly critical at interrupt-time because the context of the current process may not cover the address space containing the data buffer that the hardware interrupt handler needs to access in order to move data.

To prevent an application from passing an unauthorized address, the device driver can use the DevHlp service VerifyAccess to validate the application's authority to access the memory. Because device drivers execute at the operating system privilege level, they have access rights to segments at all privilege levels. However, a well-balanced device driver must not allow an application to force the

device driver into accessing segments which the application does not own. This check applies to addresses that an application passes within a generic IOCtl request; the OS/2 kernel validates addresses for READ and WRITE requests. If an application passes a bad address to the device driver, the device driver could halt the system if it does not verify the caller's access authority. Once an address has been verified, the device driver can proceed with the I/O request.

The DevHlp services LOCK and UNLOCK are used to fix in place a segment, which prevents the segment from being moved or swapped while the device driver needs access to it. The device driver does not need to lock segments for the READ or WRITE request packets. However, segments referenced in the generic IOCtl request packet will need to be locked by the device driver if it intends to access them at interrupt time.

Once a segment has been locked, the device driver can convert the virtual address (segment/selector:offset) into a physical address with the DevHlp VirtToPhys for later use at interrupt time.

The DevHlps AllocPhys and FreePhys allow the device driver to get and free a fixed amount of memory. The device driver must use the DevHlps PhysToVirt and UnPhysToVirt to obtain a virtual address (segment/selector:offset) to access the memory.

The device driver should choose to locate critical data structures or data transfer areas in its data segment. This is optimal for performance when access to the structures or buffers must take place at both task time and interrupt time.

Semaphore Management

There are two kinds of semaphores, RAM semaphores and system semaphores. RAM semaphores are defined by the semaphore user by allocating a DWORD variable and using the address in place of the handle in DevHlp semaphore services. OS/2 provides no resource management on RAM semaphores (such as releasing the semaphore when the owner terminates). System semaphores are created by an application through a dynamic link function call. OS/2 provides full resource management on system semaphores, including releasing of the semaphore and notification when the owner of the semaphore terminates.

Typically, a device driver creates and uses RAM semaphores to control operations among its components. They are not restricted for use by mode. The device driver may use RAM semaphores while executing in user mode.

System semaphores are typically used by a device driver to communicate to an application process. A device driver cannot create a system semaphore; although it can use the system semaphore that the application process has created. The application process must pass the application's handle to the device driver in a generic IOCtl. The device driver then uses the DevHlp service SemHandle to obtain a semaphore handle that the device driver can use. The device driver must indicate in the SemHandle call that the system semaphore is IN-USE by the device driver. When the device driver no longer needs to use the system semaphore to communicate with the application, it must call the DevHlp SemHandle and specify that the system semaphore is NOT-IN-USE.

Character Queue Management

Character queues are used by character device drivers to buffer data. The two most frequently used structures for character buffers are the FIFO and the CIRCULAR buffer.

A character device driver may use the DevHlp services to manage a simple circular buffer for characters. The DevHlp services operate on the following character queue header.

```
CharQueue STRUC
        Qsize   DW      ?       ; Size of buffer in bytes
        Qchrout DW      ?       ; Index to next char out
        Qcount  DW      ?       ; Count of characters in buffer
        Qbase   DB      ?       ; Start of buffer
CharQueue ENDS
```

Prior to using the character queue DevHlp services, a device driver must allocate the queue header and initialize the Qsize field. The DevHlp QueueInit must be called before calling any of the other character queue DevHlps. The other fields in the queue header are managed by the character queue DevHlps and do not need to be examined or altered by the device driver.

A character device driver is not required to use the character queue DevHlp services. A character device driver can define its own char-

acter buffer management, tailored to the requirements of its buffer structure.

Hardware Interrupt Management

The device driver's hardware interrupt handler is the component of the device driver which deals with a hardware interrupt. The hardware interrupt handler is called by the OS/2 kernel when the hardware interrupt occurs, therefore it must follow the FAR CALL/RET model. By convention, the hardware interrupt handler does not need to save and restore any registers it uses. This is done by the kernel. For performance reasons, the hardware interrupt handler will be called in the CPU mode that the hardware interrupt occurred. Therefore, the Hardware Interrupt Handler must not have a dependency on the mode in which it is invoked.

Before the hardware interrupt handler can be invoked, its entry point must be registered for a specific hardware interrupt. This may be done during or after device driver initialization with the DevHlp service SetIRQ. Once the call to the DevHlp has been made, the hardware interrupt handler can be invoked.

Hardware interrupt sharing is not supported on the Personal Computer AT or the Personal Computer XT Model 286 A hardware interrupt level (IRQ) that is shared among two or more devices is referred to as a shared interrupt. Interrupt sharing is an extension of a device's design. A single interrupt level (IRQ) can be shared among two or more devices if the devices are specifically built for interrupt sharing. A device driver cannot share the hardware interrupt level without cooperation from its device. This is true for both edge-triggered and level-sensitive interrupt environments.

In an edge-triggered interrupt environment, an interrupt request will be recognized by the 8259 Programmable Interrupt Controller (PIC) as a particular edge transition (like low-to-high) on the hardware interrupt request line. The interrupt request line can remain level without generating another interrupt. The 8259 PIC requires an End-Of-Interrupt (EOI) and another of the same kind of edge transition to recognize an interrupt on that interrupt level.

In a level-sensitive interrupt environment, an interrupt request will be recognized by the 8259 PIC as a particular level on the hardware interrupt request line. The interrupt condition must be removed

before the EOI is issued or else the 8259 will continue to generate interrupts for that interrupt level.

Note: OS/2 supports interrupt sharing only on the PS/2, which provides a level-sensitive interrupt environment, where multiple device drivers (devices) may share a particular hardware interrupt. On both the Personal Computer AT and the Personal Computer XT Model 286, hardware interrupts cannot be shared among multiple device drivers (devices).

The basic model for managing a hardware interrupt follows:

1. The device driver must register an interrupt handler for a hardware interrupt, specifying whether the device driver intends to share the interrupt level.
2. When invoked, the device driver interrupt handler tests the device to see if it generated the interrupt.
3. If the device has an interrupt pending or caused a spurious interrupt, the interrupt handler owns the processing of the interrupt.

   The interrupt handler services the device, resets the interrupting condition at the device, issues the End-Of-Interrupt (EOI) with the DevHlp service EOI, and RET FAR with the indicator that it owned the interrupt (CF = 0).
4. If the device does not have an interrupt pending, the interrupt handler does not own processing of the interrupt.

   The interrupt handler must RET FAR with the indicator that it does not own the interrupt (CF = 1).

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

1. Interrupt Level Sharing

   All interrupt levels have the potential to be shared. There are some restrictions.

   SYSTEM TIMER RULE The system timer interrupt level (IRQ 0) cannot be shared.

   The system timer interrupt level is owned by a DOS mode interrupt handler for compatibility operations.

ILL-BEHAVED DEVICE RULE An interrupt handler for an ill-behaved device must not share an interrupt level.

An ill-behaved device is one that generates interrupts before its interrupt handler is installed or is one that cannot be told to stop generating interrupts.

Well-behaved devices are devices that do not power-up with interrupts pending and do not remain active after their handlers have terminated. Also, well-behaved devices do not usually generate spurious interrupts.

BIOS INT HANDLER RULE A bimodal interrupt handler that uses the BIOS interrupt handler to support interrupt processing from I/O generated by the DOS mode must not share an interrupt level.

A BIOS interrupt handler does not share its interrupt level, but assumes that it owns the interrupt processing.

2. Initializing the Interrupt Vector

SET IRQ RULE The device driver must indicate when signing up for a hardware interrupt level with the DevHlp SetIRQ that it will share the interrupt.

IRQ ENFORCEMENT RULE If a device driver signs up for a hardware interrupt indicating that it will not share it, then a subsequent SetIRQ request to share that hardware interrupt will be refused. Conversely, if a device driver signs up for a hardware interrupt indicating that it will share it, then a subsequent SetIRQ request to exclusively own the hardware interrupt will be refused.

DOS MODE SHARING RULE Interrupt sharing cannot be performed by a DOS mode interrupt handler.

A hardware interrupt is owned either by one or more bimodal interrupt handlers (OS/2 device drivers) or by a single DOS mode interrupt handler.

A DOS mode interrupt handler exclusively owns its hardware interrupt. It may not share its interrupt with a bimodal device driver. A DOS mode interrupt handler may not share its interrupt because, as part of the DOS mode, it can execute only when the DOS mode is in the foreground.

COROLLARY If a bimodal device driver (OS/2 device driver) owns a device that is accessible from a BIOS software interrupt, the bimodal device driver's interrupt handler will own the hardware interrupt level, not BIOS. However, the bimodal device driver may NOT share its interrupt level with other bimodal device drivers, if the bimodal interrupt handler uses the BIOS interrupt handler when processing an interrupt generated by an I/O request from the DOS mode.

The bimodal device driver's interrupt handler may need to support DOS mode I/O by using the BIOS interrupt handler in the DOS mode. In this case, the bimodal device driver must be aware that the BIOS interrupt processing does not include a check for ownership of the interrupt level. In addition, the bimodal device driver must be aware of the background processing of OS/2 mode processes while the DOS mode is in the foreground — the bimodal device driver is always called in the current processor mode of operation.

IRQ MASK RULE The operating system owns the masking of the hardware interrupt at the 8259 interrupt controller.

The operating system will enable the hardware interrupt at the 8259 interrupt controller when the first interrupt handler signs up for the hardware interrupt. This permits the interrupt handler to communicate with its device during initialization.

3. Processing The Interrupt

STI ENTRY RULE Interrupt handlers that share interrupts will be entered with processor interrupts enabled.

This is to prevent the lockout of higher priority hardware interrupts, because the search for the owner of the current interrupt level takes a variable amount of time.

Interrupt handlers that do not share interrupts will be entered with processor interrupts disabled.

A DOS mode interrupt handler will be entered with processor interrupts disabled for compatibility.

IRQ OWNERSHIP RULE The device driver interrupt handler, when invoked, must always interrogate its device to see if its device caused the interrupt. If the interrupt handler's device caused the interrupt, then the interrupt handler owns the processing of the interrupt.

If the interrupt handler owns the processing of the interrupt, it may briefly disable processor interrupts for critical operations. It must issue the EOI as soon as possible.

The device driver interrupt handler must be aware that once it issues the EOI, it could be reentered at its interrupt handler's entry point.

If the interrupt handler's device did not cause the interrupt, then the interrupt handler must not issue an EOI.

INT RETURN RULE The interrupt handler, after taking the appropriate action in processing the interrupt, must return an indication whether it claimed the interrupt or not.

If the interrupt handler owns the interrupt, then 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, then it must set the CARRY FLAG (CF = 1) and issue a FAR RET.

SEARCH RULE The operating system calls each interrupt handler registered for a particular interrupt level until one of the interrupt handlers claims the interrupt.

EOI RULE Management of the 8259 interrupt controllers is the responsibility of the operating system. However, the End-Of-Interrupt (EOI) is the responsibility of the interrupt handler.

The interrupt handler must use the DevHlp EOI service to issue the EOI as soon as possible in the processing of its interrupt. This permits the 8259 interrupt controller to process other interrupt requests at the current interrupt priority as well as interrupt requests of lower priorities.

In a level-sensitive interrupt environment, the EOI must not be issued to the 8259 interrupt controller(s) until the interrupt condition at the device is removed.

Advanced BIOS requires that all ABIOS staged-on interrupt request blocks be processed for the LID that owns the interrupt prior to the EOI. (Refer to the ABIOS EOI Placement Rule.)

PhysToVirt RULE Selectors used for PhysToVirt represent a critical resource; an interrupt handler that uses PhysToVirt must not issue the EOI until after it no longer needs the addresses generated by PhysToVirt. Otherwise, the interrupt handler should disable processor interrupts before issuing the EOI; this will allow the interrupt handler to use the temporary selectors for its interrupt level without getting another interrupt on its level.

POSITION RULE An interrupt handler that shares an interrupt level must not depend on its position in the list of handlers for that interrupt level.

4. Advanced BIOS Considerations For Interrupt Processing

ABIOS REQUEST BLOCK RULE The interrupt handler for a particular Logical ID (LID), when invoked by the operating system, must call Advanced BIOS for each ABIOS request block that is Incomplete-Waiting-On-Interrupt, even if one of the request blocks gets the return indicator that the interrupt belongs to it.

ABIOS EOI PLACEMENT RULE The EOI must be issued after all ABIOS staged-on interrupt request blocks have been processed for the LID that owns the interrupt.

ABIOS LID IRQ RULE Advanced BIOS defines one and only one interrupt level per LID.

If a device driver handles more than one LID on the same interrupt level, then the device driver could choose to register only one interrupt handler for any LID on that level. In this case, the operating system will call the interrupt handler only once when the interrupt occurs; the interrupt handler must manage the processing of more than one LID in order to determine if it owns the interrupt processing for them.

In this case, the device driver interrupt handler should be aware of the Fairness Criteria problem. A LID at the end of the interrupt handler's list will not get as much service as a very active LID at the front of the list.

5. Spurious Interrupts

   In an edge-triggered interrupt environment, to handle a spurious interrupt, reset the interrupt at the 8259 interrupt controller (EOI), issue the global rearm if sharing interrupts, and enable processor interrupts. Generally, these actions would be taken by the last interrupt handler in the list of interrupt handlers.

   In a level-sensitive interrupt environment, to handle a spurious interrupt, reset the interrupt condition at the device, issue the EOI, and enable processor interrupts.

   Advanced BIOS provides the capability to reset a spurious interrupt at the device through the use of an Advanced BIOS Default Interrupt Handler for the Logical ID (LID). The interrupt handler calls the Advanced BIOS Default Interrupt Handler for its LID if there are no outstanding Incomplete-Waiting-on-Interrupt request blocks. The Advanced BIOS Default Interrupt Handler will indicate either that the interrupt condition was successfully reset or that the interrupt did not belong to the device referenced by the LID. In the case that the Advanced BIOS Default Interrupt Handler replies that the device was successfully reset then the interrupt handler must issue the EOI and return as owning the interrupt.

   For the case where there are Incomplete-Waiting-On-Interrupt request blocks outstanding, Advanced BIOS keeps track of which interrupts are expected, and will automatically service a spurious interrupt when called by the interrupt handler. The interrupt handler must call Advanced BIOS with each and every request block that is Incomplete-Waiting-On-Interrupt for a LID even if the first one returns an indication that it performed some processing. The interrupt handler must be able to process the spurious-interrupt return code from any one of these calls to the Advanced BIOS Interrupt service.

   For a device driver that directly interfaces to the device, it must check the device for the interrupt condition, even if the interrupt condition does not correspond to an outstanding I/O request. If the device had caused the spurious interrupt, the interrupt handler must reset the interrupting condition at the device, issue the EOI, and return as owning the interrupt.

   Note: If the device causing the spurious interrupt in the level-sensitive interrupt environment is not identified and reset, then

the interrupt level is locked up. This is a feature of the 8259 interrupt controller operating in level-sensitive mode.

6. DEINSTALL Considerations

   Refer to the section, "DEINSTALL Considerations" on page 7-64, for details.

Notes: A single device driver with a single interrupt handler for a particular interrupt level may share its interrupt level among one or more devices that it owns. This case applies to devices of similar or same nature, for example, a printer device driver supporting more than one printer (adapter) on one interrupt level. For this case, the operating system will invoke the device driver's interrupt handler when the hardware interrupt occurs. The device driver's interrupt handler must determine which of its devices caused the interrupt; the interrupt handler will not get called for each device it manages.

If the device driver is supporting multiple devices on the same hardware interrupt, it only needs to process the first device it discovers that was causing an interrupt on the interrupt level in question.

Because of the level-sensitive interrupt environment, other devices that are requesting service will cause another interrupt to be generated and the device driver would be re-entered at the same entry point. Because of this, the device driver should strategically place the EOI to allow an orderly processing of any re-entrant interrupt requests.

However, if the device driver registers a separate unique interrupt handler entry point for each device it owns, with the interrupt handlers sharing the same interrupt level, then the device driver must adhere to the interrupt sharing rules. The operating system will invoke each interrupt handler, until one handler claims the interrupt. To the operating system, each registered interrupt handler entry point appears as a separate interrupt handler. This allows the device driver's interrupt handler(s) to be called for each device.

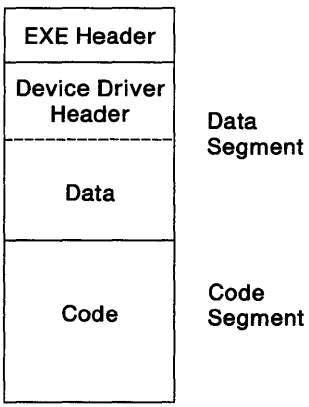
Device Driver Program Model

The program model for the OS/2 device driver is a small model. In other words, the device driver consists of two segments, a single code segment and a single data segment. The device driver executable image does not contain a stack segment; a stack is provided by the system.

Because the device driver has only one code segment, it must not have any FAR CALLs or FAR JMPs internal to the code segment. The device driver does not need to examine the code segment value. OS/2 tracks the segment/selector and sets the proper value for the CPU mode in the CS register when it calls the device driver components. This applies to both task time and interrupt time. OS/2 tracks the data segment value in a similar fashion, and sets the proper value for the CPU mode in the DS register.

The device driver loadable image may contain extra space in its data segment area to be referenced and extra code in its code segment area to be executed at initialization. Once the device driver has completed initialization, only the primary areas of the code and data segments will be kept. The extra space which is no longer needed, will be returned to the system.

The file image of the device driver follows:



The data segment must be the first segment after the .EXE file header. This allows the device header to be located immediately

after the .EXE file header because the device header is required to be located at the beginning of the file.

Device Driver Header

The device driver's data segment must contain a Device Header as the very first item. The Device Header has the following structure.

Field	Length
Pointer To Next Header	DWORD
Device Attribute	WORD
Offset To Strategy Routine	WORD
Reserved	WORD
Name or Units	8 BYTES
Reserved	8 BYTES

Pointer to Next Device Header Field

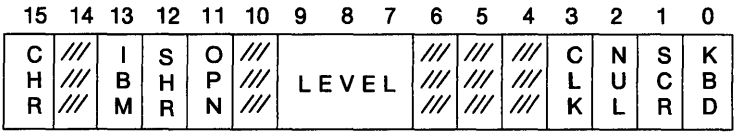
The pointer to the next header is set by OS/2 at the time the device driver is loaded. For loadable device drivers, this field should be set to -1. The pointer-to-next-header field is set by SysInit and should be left blank.

Note: For a character device driver that supports multiple devices, the data segment contains a device driver header for each device. These headers must be linked together and the first header must be set to -1. The first word is an offset and the second word is the segment.

Device Attribute Field

The Device Attribute field describes the characteristics of the device driver to the system.

The format of the OS/2 device attribute field is:



The attributes are:

Bit#	***Meaning***
15	Set if character device driver
14	Reserved = 0
13	Set if non-IBM block format, (block device drivers only). Set for output-until-busy support, (character device drivers only).
12	Set if support shared device access checking (character devices)
11	Set if support removable media (block devices) or device open/close (character devices)
10	Reserved = 0
9-7	Function level where 001 = OS/2 device driver
6	Reserved = 0
5	Reserved = 0
4	Reserved = 0
3	Set if CLOCK device
2	Set if NULL device
1	Set if standard output device (STDOUT)
0	Set if standard input device (STDIN)

Bit 15 Bit 15 is the device type bit. Use bit 15 to tell the system if the device driver is a block or character device. For block device drivers, bit 15 is 0. For character device drivers bit 15 is 1.

Bit 13 For block device drivers, bit 13 indicates the method the driver uses to determine the media type.

If a block device driver uses information in the BPB to determine the media type, bit 13 should be set to 1. If the device driver uses the media descriptor byte to determine the media type, bit 13 should be 0.

Bit 12 Bit 12 is the shared bit. It is set if the device name is NOT to be protected by sharer. (Has no meaning for block device drivers, must be 0.)

If clear (default), file system sharing rules DO NOT apply to the device, and it is the responsibility of the device driver to provide contention control.

If set, file system sharing rules DO apply to the device, just like they apply to any other file system name. In addition, any given physical device may have only one logical name. (Devices cannot have aliases.)

Bit 11 For block device drivers, bit 11 is the removable media bit. If set, this bit indicates that the device driver handles removable media.

For character device drivers, bit 11 is the open/close bit. If set, this bit indicates that the device driver must receive OPEN AND CLOSE request packets.

Bit 9-7 Bits 9-7 indicate the function level where 001 = OS/2 device driver.

Bit 3 Bit 3 is the clock device bit. It is used by a character device driver to indicate the system clock device.

Bit 2 Bit 2 is the NULL attribute bit. It is used by character devices only. Use bit 2 to tell OS/2 if your character device driver is a NULL device. Although there is a NULL device attribute bit, you cannot reassign the NULL device. This is an attribute that exists for OS/2 so that OS/2 can tell if the NULL device is being used.

Bits 1 and 0 For character devices, bits 1 and 0 are the standard input/standard output bits. Use these bits to tell OS/2 if your character device driver is the new standard input or standard output device.

Offset to Strategy Routine Field

The offset to the strategy routine field contains the offset from the start of the code segment to the strategy entry point. OS/2 uses this offset to call the strategy routine. The pointer is a word value contained in the device header.

Name/Units Field

The name/units field contains the name of a character device supported by the character device driver or the number of units supported by the block device driver.

For a character device driver, the name of the device must be ASCII characters and must be left-justified with the remaining space set to blanks. The device name is used by applications to identify the device for I/O. A character device driver should consider the following rule when selecting a device driver name.

- **Rule**

  A device name takes precedence over a filename in a DosOpen function call. This means that files cannot have the same name as a character device. The DosOpen function call will always open the device rather than the file.

  Note: To avoid such conflicts with filenames, a character device driver should choose a character string with some unusual character such as a $ sign.

For a block device driver, the number of units can be placed in the first byte. This is optional because OS/2 will fill this field during device driver initialization.

For character device drivers using ABIOS, the device name represents a single device identified by the Logical ID (LID). For block device drivers using ABIOS, the number of units is equivalent to the number of devices (or units) under the LID.

Creating a Device Driver

To create a device driver that OS/2 can install, perform the following:

- Originate the device header at 0, not at 100H.
- Set up the device header fields.
- Link the segments as a library.

Note: Because OS/2 installs the driver anywhere in memory, care must be taken in any memory references. You should not expect that your driver will always be loaded at the same place every time.

Device Driver Initialization

Device driver initialization occurs during system initialization. During system initialization, base device drivers are preloaded with the operating system. Installable device drivers are loaded during CONFIG.SYS processing, via the DEVICE= configuration command. After a device driver has been loaded, it will be called at its strategy routine entry point with the request packet for the INIT command.

OS/2 device drivers have the following characteristics at INIT-time:

- Occupy memory below 640 Kb.
- Initialize in the OS/2 mode.
- Are able to use Advanced BIOS services at INIT-time.
- Have IOPL at all times.
- Are able to service hardware interrupts.
- Are able to use timer services.
- Are able to use some OS/2 dynamic link function calls.

During CONFIG.SYS processing, each DEVICE= command is processed on a first-come-first-serve basis:

- The DEVICE= device driver file image is loaded into low memory.
- The check for DEINSTALL of a previously initialized device driver is performed.
- If DEINSTALL is successful, then the newly loaded device driver is initialized.

Installable OS/2 device drivers initialize in the OS/2 mode. The device driver strategy routine will run under the thread of the System Initialization Process, at application level, with I/O privilege. Because of the special system process, the installable device driver is allowed to make dynamic link function calls only at INIT-time.

Device Driver INIT-Time Function Call Summary

For function call details, refer to *Technical Reference, Vol. 2.*

The list of function calls that the device driver can make is as follows:

DosBeep	Generate Sound From Speaker
DosCaseMap	Perform Case Mapping
DosChgFilePtr	Change (Move) File Read/Write Pointer
DosClose	Close File Handle
DosDelete	Delete File
DosDevConfig	Get Device Configuration
DosDevIOCtl	I/O Control for Devices
DosFindClose	Close Find Handle
DosFindFirst	Find First Matching File
DosFindNext	Find Next Matching File
DosGetCtryInfo	Get Country Information
DosGetDBCSEv	Get DBCS Environmental Vector
DosGetEnv	Get Address of Process Environment String

DosGetMessage	System Message with Variable Text
DosOpen	Open File
DosPutMessage	Output Message Text to Indicated Handle
DosQCurDir	Query Current Directory
DosQCurDisk	Query Current Disk
DosQFileInfo	Query File Information
DosQFileMode	Query File Mode
DosRead	Read from File
DosWrite	Synchronous Write to File

DevHlp services are also available to device drivers at INIT-time.

Note: Because device driver initialization is invoked by way of the strategy routine, the device driver must not issue a DosExit function call. Instead, the device driver should return the INIT request packet by setting the packet's return status and performing a FAR RET to the kernel.

For more information see "0H / INIT Initialize Device" on page 7-43.

Replacing Character Device Drivers

OS/2 character device drivers can be replaced. For system character device drivers, the appropriate bits in the attribute field of the device driver header and the name of the character device driver must match.

When the new device driver is loaded, the attribute field and name are used to determine if the new device driver is attempting to replace a driver already installed. If so, the previously installed device driver is requested to DEINSTALL the indicated device. If the already-installed device driver refuses the DEINSTALL command, the new device driver is not allowed to initialize. If the already-installed device driver performs the DEINSTALL, the new device driver is initialized.

Note: The DEINSTALL command is needed to allow a device driver to relinquish its interrupt vectors and its allocated physical memory.

Compatibility with Previous-Level Device Drivers

The term "previous-level" is used here to indicate DOS levels below DOS 3.3

Not all previous-level DOS device drivers can be allowed to run in the DOS mode. The supported set of previous-level device drivers has the following restrictions:

- Previous-level block device drivers are not permitted in the DOS mode. Block device drivers must be written to the OS/2 interfaces.
- Only a limited set of previous-level character device drivers can be supported by OS/2. In order to run in the DOS mode, a previous-level character device driver must conform to the following rules:
  - The character device driver cannot have a hardware interrupt handler; its device must be a polled device rather than an interrupt-driven one.
  - The character device driver can be called by OS/2 in the DOS mode with all of the packets supported by character devices:

    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
    13 — DEVICE OPEN
    14 — DEVICE CLOSE
    16 — GENERIC IOCtl

    Receipt of these packets is limited by the same requirements on the attribute field of the device header as in DOS 3.3. For example: IOCtl bit in the device header must be 1 to receive IOCtl requests.

The side effect of running a previous-level character device driver is that its device may be used only in the DOS mode. Only applications running in the DOS mode can perform I/O to this device. Applications in the OS/2 mode are not allowed to access the device.

Certain devices cannot be exclusive to the DOS mode. Character devices in this category include:

- Mouse
- Clock

Consequently, a previous-level clock device driver cannot be supported.

Initialization of Previous-Level Device Drivers

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

Previous-level device drivers are loaded and initialized in DOS mode. The rules for replacing previous-level character device drivers are the same. 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 DOS 3.3. No INT 21H functions can be performed from the device driver initialization code.

DOS Execution Environment Generic IOCtl Support

There are two types of generic IOCtls supported in the DOS mode.

- Function: AL=0DH

  Where this function is the same as DOS 3.3 with the addition that the register pair SI:DI is the address of the parameter block in OS/2 and DS:DX is the address of the data packet.

- Function: AL=0CH

  This function is similar to function AL=0DH except that BX contains a handle to a device instead of a drive letter. This function is useful for character devices.

The register contents are as follows:

Register	*Contents*
AH	44H — IOCtl request
AL	0DH — Drive oriented
	0CH — Handle oriented
BL	Drive number
BX	Handle value
CH	Category
CL	Function
DS:DX	Data block
SI:DI	Parameter block

Refer to the specific device IOCtl descriptions for the Categories and Functions supported.

DOS Execution Environment Software Interrupt Support

The following is a list of the software interrupts supported and the compatibility exceptions for DOS mode operation:

Interrupt		*Comments*
05H	Print screen	Request ignored **Note:** Shift-Print screen works for text mode screens in both the OS/2 and DOS modes.
12H	Memory size	Supported — size limited to DOS mode size
13H	Disk / Diskette	For non-removable media only — these functions are supported: 01H — read status 02H — read sectors 0AH — read long 15H — read DASD type

14H	ASYNC	If the ASYNC device driver is loaded in the system, then INT 14H will not function for its related ASYNC ports unless the utility SETCOM40 is used. See SETCOM40 in the *User's Reference* for INT 14H considerations.
15H	Misc	Functions not supported: 87H — Block Move 88H — Extended memory size 89H — Virtual mode 90H — Device busy 91H — Int. complete Functions supported with restrictions: 83H — Event wait 86H — Wait **Note:** Both 83H and 86H are supported; but the timer granularity is on the order of 31ms. Because the RTC (Real Time Clock) is free running, there will be a variance of up to 1 RTC tick.
17H	Printer	Supported by OS/2 device driver
19H	Reboot (Re-start)	Supported — However, this does not operate in the same manner as DOS 3.3. The system is restarted as if Ctrl-Alt-Del was pressed.
1AH	TOD	Functions not supported: 02H — Read RTC time 03H — Set RTC time 04H — Read RTC date 05H — Set RTC date 06H — Set RTC alarm 07H — Reset RTC alarm

1EH	Diskette parameters	Not used by device driver after boot process
24H	Hard error	Supported — OS/2 calls the application when a hard error occurs.
26H	Direct write	An error is returned on requests for non-removable media.
2FH	Multiplex	Returns error "Not installed, not to be installed" (printer only — request AL=0 returns AL=1);
33H	Mouse	Supported — When the OS/2 Mouse device driver is loaded, INT 33H functions are available.

Using Advanced BIOS

There are two methods that device drivers may use to invoke Advanced BIOS, the Advanced BIOS Transfer Convention and the Operating System Transfer Convention. For the Advanced BIOS Transfer Convention, the Advanced BIOS Common Entry Points are invoked to locate the specific Start, Interrupt, or Timeout entry points for the requesting Logical ID. For the Operating System Transfer Convention, the specific Start, Interrupt, or Timeout entry points must be located for the requesting Logical ID and called.

OS/2 internal device drivers use the Operating System Transfer Convention to invoke Advanced BIOS services. User-written, installable, device drivers may use either the Advanced BIOS Transfer Convention or the Operating System Transfer Convention. DevHlps are provided for both calling conventions.

Note: Both kinds of device drivers mentioned above will be commonly termed as ABIOS device drivers.

For performance reasons, ABIOS device drivers should call Advanced BIOS services with processor interrupts enabled.

Device Driver Data Segment

OS/2 recommends that an ABIOS device driver use its data segment to contain the Advanced BIOS request blocks and data transfer buffers. The device driver data segment is located in low memory and the operating system guarantees addressability to the data segment regardless of the processor mode (protect or real). The device driver may also assume that the physical location of the device driver data segment will not move. This will allow physical data transfers to take place to buffers within the device driver's data segment.

By using its data segment, the device driver can create logical addressability to these data areas (for Advanced BIOS) in a mode-independent manner and without interrupt disable time considerations. Physical data transfers to buffers outside of the device driver's data segment may take place if the buffer is locked.

Obtaining a Logical ID

During its initialization, an ABIOS device driver must obtain the Logical ID (LID) for its physical device.

The allocation of a LID is managed by the operating system. This ensures that the device driver gets a unique LID for its device type. The operating system provides a DevHlp function GetLIDEntry to obtain the LID for a device driver.

The DevHlp GetLIDEntry finds the LID for the specified Device ID and allocates it to the calling device driver.

The counterpart to GetLIDEntry is the DevHlp FreeLIDEntry. This service is required when the device driver DEINSTALLs or terminates to release the device driver's claim to the LID. Refer to the section "DEINSTALL Considerations" on page 7-64 for more details.

The operating system will prohibit access to a certain LID; specifically, the LID for System Services. The operating system management of LID access is similar to the management of hardware interrupt levels or I/O ports.

Calling Advanced BIOS Services

For ABIOS device drivers that use the Operating System Transfer Convention, a DevHlp service (ABIOSCall) is provided to invoke Advanced BIOS with the mode specific correct set of parameters. The device driver passes the ABIOS request block pointer, its LID, and the ABIOS primary function (start, interrupt, or timeout) to the DevHlp ABIOSCall. This sets up the stack for the call to Advanced BIOS and calls the ABIOS function.

For ABIOS device drivers that use the Advanced BIOS Transfer Convention, a DevHlp service (ABIOSCommonEntry) is provided to invoke the Advanced BIOS Common Entry Points The device driver passes the mode specific pointer to the ABIOS request block and the ABIOS primary function (common start, common interrupt, or common timeout) to the DevHlp service. The DevHlp service sets up the stack, and calls the requested ABIOS common entry point.

Note: The return code of the ABIOS function will be in the ABIOS request block.

Mapping Device Names to LID

Having identified its LID and the number of devices or physical units the LID represents, the device driver must map each of its device names to a unit within that LID.

Note: A device driver supports all units under a given LID.

All device drivers are known to the operating system by device names, whether these names correspond to the ASCII string device name in the header for character device drivers or to the logical units (which correspond to drive letters) in the header for block device drivers.

In the case of a character device driver with a single device driver header, its device name must be mapped to the first unit of the LID it obtained. If the character device driver has one device header but its LID had multiple units, then the rest of the units are not used.

In the case of a character device driver with multiple device driver headers, the operating system will call the strategy routine entry point for each header during device driver initialization.

1. The first entry point called must map its device name to the first unit of the LID.
2. The second entry point called must map its device name to the next unit of the LID.
3. If the LID that was obtained by the first entry point has only one unit, the second entry point must obtain another LID and map its device name to the first unit of the second LID.
4. The device driver must start with the first LID and consume all the units before going to the next LID.

For example:

The printer device driver has three device headers (LPT1, LPT2, and LPT3), respectively. The first entry point will map LPT1 to the first unit of the LID it obtained (let's use LID #12). If LID #12 supports only one unit, the second entry point will map LPT2 to the first unit of another LID it must obtain (let's use LID #17). If LID #17 supports another unit, the third entry point will map LPT3 to the second unit of LID #17.

In the case of a block device driver, the block device driver must obtain the necessary number of LID/units for the number of logical units it supports. The block device driver maps the first logical unit to the first-LID/first-unit, the second logical unit to the next available LID/unit, and so forth.

Handling ABIOS Requests

Refer to "Notes On Writing a Device Driver using Advanced BIOS" on page 7-72, for a discussion on handling requests to Advanced BIOS.

A device driver must assume that it owns all outstanding ABIOS request blocks for a given Logical ID. During interrupt-time proc-

essing, the device driver must call Advanced BIOS for each outstanding request that is Incomplete-Waiting-On-Interrupt.

Note: This in one of the reasons that a Logical ID is not shared among device drivers.

Request Packets

The device driver strategy routine is called with ES:BX pointing to the request packet. The pointer to the request packet (ES:BX) is bimodal. In other words, the pointer is valid in both the DOS mode and the OS/2 mode.

OS/2 does not guarantee that the order of API requests that are issued by multiple threads will be preserved in the ordering that the corresponding request packets arrive at the device driver. Multiple application threads or threads created due to DosReadAsync and DosWriteAsync can get blocked in the operating system. This allows a device driver request packet for an API request by a subsequent thread that does not get blocked to arrive out of order. A device driver is responsible for providing a synchronization mechanism between itself and application processes if it supports multiple outstanding requests; also, request packet ordering must be preserved.

The request packet consists of two parts, the request header and the command-specific data field. The format of the request packet is detailed below.

Field	Length
Length of Request Packet	BYTE
Block Device Unit Code	BYTE
Command Code	BYTE
Status	WORD
Reserved	DWORD
Queue Linkage	DWORD
Command-specific Data	BYTES

Length of Request Packet Field

The length of the request packet is set to the total length in bytes of the request packet (the length of the request header plus the length of the data).

Block Device Unit Code Field

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

Command Code Field

The command code indicates the requested function. The command codes are listed in the following summary.

Summary of Commands for Devices

CODE	FUNCTION	BLOCK	CHAR
0H	INIT	*	*
1H	MEDIA CHECK	*	
2H	BUILD BPB	*	
3H	Reserved		
4H	READ (input)	*	*
5H	NONDESTRUCTIVE READ NO WAIT		*
6H	INPUT STATUS		*
7H	INPUT FLUSH		*
8H	WRITE (output)	*	*
9H	WRITE WITH VERIFY	*	*
AH	OUTPUT STATUS		*
BH	OUTPUT FLUSH		*
CH	Reserved		
DH	DEVICE OPEN	*	*
EH	DEVICE CLOSE	*	*
FH	REMOVABLE MEDIA	*	
10H	GENERIC IOCtl	*	*
11H	RESET MEDIA	*	
12H	GET LOGICAL DRIVE MAP	*	
13H	SET LOGICAL DRIVE MAP	*	
14H	DEINSTALL		*
15H	Reserved		
16H	PARTITIONABLE FIXED DISKS	*	
17H	GET FIXED DISK/LOGICAL UNIT MAP	*	
18H	Reserved		
19H	Reserved		
1AH	Reserved		

The commands are described in detail in the command section of this chapter.

Request Packet Status Field

On entry to the strategy routine, the status field is only defined for Open and Close request packets. For all other request packets the status field is undefined on entry.

For an Open request packet, Bit 3 (08H) of the status field is SET if the packet was generated from a DosMonOpen, otherwise it was a DosOpen.

For a Close request packet, Bit 3 (08H) of the status field is SET if the packet was generated by a DosMonClose or a DosClose of a handle that was generated by a DosMonOpen (so that monitor handles generated that are left open when a process exits will be closed properly). Otherwise, it was a DosClose on a non-monitor handle.

On exit from the strategy routine the status field describes the resulting state of the request as shown below:

15	14	13-10	9	8	7-0
ERROR	DEV ERROR	RESERVED	BUSY	DONE	ERROR CODE (bit 15 on)

Bit 15 is the Error bit. If this bit is set, the low 8 bits of the status word (7-0) indicate the error code.

Note: If the category is user-defined, then the error returned to the caller is FF00h ANDed with the byte-wide error code.

Bit 14 is a device driver defined error if set in conjunction with bit 15.

Note: If the category is user-defined, then the error returned to the caller is FE00h ANDed with the byte-wide error code.

Bits 13 - 10 are reserved.

Bit 9 is the Busy bit. It is only set by status calls and the removable media call. See "STATUS" and "REMOVABLE MEDIA" in this chapter for more information about the calls.

Bit 8 is the Done bit. If it is set, it means the operation is complete. The driver sets the done bit to 1 when it exits.

Bits 7-0 are the low 8 bits of the status word. If bit 15 is set, bits 7-0 contain the error code. The error codes and errors are:

Error Codes	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

Uncertain Media (10H) should be returned when the state of the media in the drive is uncertain. This response should NOT be returned to the INIT command. For fixed disks, the device driver must begin in a Media Uncertain state in order to have the media correctly labelled. In general, the following guidelines may be used to determine when to respond with uncertain media.

- When a drive-not-ready condition is detected. (In this case, return uncertain media to all subsequent commands until a reset media command 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.

Character I/O call interrupted (11H) should be returned when the thread performing the I/O was interrupted out of a DevHlp Block before completing the requested operation.

Monitors not supported (12H) should be returned for monitor commands (monitor open/close, register IOCtl) if monitors are not supported by the device driver.

Invalid parameter (13H) should be returned when one or more fields of the request packet contain invalid values.

Queue Linkage Field

The queue linkage is provided to maintain a linked list of request packets. The device driver may use the request queue management DevHlp services, or it may use its own queue management.

Note: Because a pointer to a request packet is bimodal (valid in both the DOS mode and the OS/2 mode), the pointer may be used directly in the queue linkage rather than a 32-bit physical address.

Command-Specific Data Field

The command code in the request header tells the device driver which function to perform.

The function and parameters of a command appear in the command-specific data area of the request packet. The commands and the actual formats of the corresponding request packets are discussed in the following sections.

Note: All DWORD pointers are stored with offset first, then segment.

Purpose

Initialize the device.

Format of Request Block

Field	Length
Request header	13 BYTES
Data_1	BYTE
Pointer_1	DWORD
Pointer_2	DWORD
Data_2	BYTE

Remarks

On entry, the request block contains the following fields as inputs to the device driver:

Pointer_1 Points to the DevHlp Entry Point

Pointer_2 Points to the INIT arguments

Data_2 Drive number for the first block device unit

The DevHlp Entry Point is a bimodal address and is valid in both the DOS mode and the OS/2 mode.
The DevHlp Entry Point is called to invoke a service specified in the DL register.

The arguments for installable device drivers from the DEVICE= line in the CONFIG.SYS file allow the device driver to use configurable parameters to initialize itself and its device.

0H / INIT
Initialize Device

At initialization time, the device driver runs as a thread under a protect mode process at application level with I/O privilege. The device driver may issue certain OS/2 dynalink function calls at this time. Refer to "Device Driver Initialization" on page 7-26 for more details.

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

Field	*OUTPUT Information for INIT success*
Data_1	Number of logical block devices or units (block devices only)
Pointer_1	WORD offset to end of code segment
	WORD offset to end of data segment
Pointer_2	Points to the BPB array for the logical block devices or units (block devices only)
Status	Set the status word in the request header to 0100H

A block device driver must return in Data_1 the number of logical devices or units that are available. The kernel's file system layer will assign sequential drive letters to these units. A character device driver will set Data_1 to zero.

Both block device drivers and character device drivers must set Pointer_1 with the offsets of the code and data segments. This allows a device driver to release code and data needed 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 (64 Kb) and let it release the amount that it does not need.

Note: Remember that the device driver code and data segments reside in memory below 640K. The DOS mode requires contiguous memory below 640K. Although memory returned by the device driver from its data segment is available to the system, it is not available for the DOS mode.

A block device driver must return an array of BPBs for the logical units that it supports in Pointer_2. A character device driver will set Pointer_2 to zero.

The Status field in the request packet header must be set to indicate no error and done.

If the device driver determines that it cannot set up the device and wants to quit, it is recommended that it return with the error bit in the request packet status field set to 1. The device driver can also return the following:

Field	***OUTPUT Information for INIT failure***
Data_1	BYTE 00H
Pointer_1	WORD 0000H
	WORD 0000H
Status	810CH

The status field in the request packet header must be set to indicate the failure of the INIT request with the General Failure error return code. The Status must also indicate that the request is done.

One of the above techniques must be used to return device initialization failures from the device driver to the system initialization process.

A character device driver that contains multiple device driver headers can fail initialization on a subset of the headers in its header chain.

The system initialization process remembers the last non-zero size code and data segment offsets returned for the devices in the device driver that completed initialization. These last values are used to resize the device driver's code and data segments after INIT packets have been sent to the device driver for each device in the device driver header chain.

0H / INIT
Initialize Device

When a device in the header chain cannot be initialized, the device driver can set the code and data segments to zero, and/or set the error bit in the request packet status field to indicate initialization failure for that device. The device driver will not receive any future request packets for a specific device if it returns a failure for the INIT request packet for that device. If none of the devices in the device driver header chain pass initialization, then the device driver will not remain loaded.

Because the system initialization process maintains the pass/fail return status for each device header in a device driver header chain, it is not recommended that the device driver manipulates the linkages of the headers.

Purpose

Determine the state of the media.

Format of Request Block

Field	Length
Request header	13 BYTES
Media descriptor	BYTE
Return code	BYTE
Return pointer to previous volume ID if supported	DWORD

Remarks

On entry, the request packet will have the media descriptor field set for the drive identified in the request packet header.

The device driver must perform the following actions for the MEDIA CHECK request:

- Set the status word in the request packet header.
- Set the return code where:
  -1 = Media has been changed
  0 = Unsure if media has been changed
  1 = Media unchanged

1H / MEDIA CHECK
Check the Media

Examples of DOS media descriptor bytes:

Disk Type	# Sides	# Sectors/ Track	Media Descriptor
Fixed disk	--	--	F8H
3 1/2-inch	2	9	F9H
3 1/2-inch	2	18	F0H
5 1/4-inch	2	15	F9H
5 1/4-inch	1	9	FCH
5 1/4-inch	2	9	FDH
5 1/4-inch	1	8	FEH
5 1/4-inch	2	8	FFH
8-inch	1	26	FEH
8-inch	2	26	FDH
8-inch	2	8	FEH

Note: To determine whether you are using a single-sided or a double-sided diskette, for 8-inch diskettes (FEH), attempt to read the second side, and if an error occurs you can assume the diskette is single-sided.

For 8-inch diskettes:

FEH (IBM 3740 Format). Single-sided, single density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 1 reserved sector, 2 File Allocation Tables (FATs), 68 directory entries, 77*26 sectors.

FDH (IBM 3740 Format). Double-sided, single density, 128 bytes per sector, soft sectored, 4 sectors per allocation unit, 4 reserved sectors, 2 FATs, 68 directory entries, 77*26*2 sectors.

FEH Double-sided, double density, 1024 bytes per sector, soft sectored, 1 sector per allocation unit, 1 reserved sector, 2 FATs, 192 directory entries, 77*8*2 sectors.

Application programmers are encouraged to use the Generic IOCtl — Get Device Parameters (Category 8, function 63) and reference the BPB (BIOS Parameter Block) to determine the type of media.

2H / BUILD BPB
Build BIOS Parameter Block

Purpose

Build the BIOS Parameter Block (BPB). This is requested when the media has changed or when the media type is uncertain.

Format of Request Block

Field	Length
Request header	13 BYTES
Media descriptor	BYTE
Transfer address	DWORD
Pointer to BPB table	DWORD
Drive number	BYTE

Remarks

On entry, the request packet will have the media descriptor set for the drive identified in the request packet header. The transfer address is a virtual address to a buffer containing the boot sector media if the block device driver attribute field has bit 13 set, otherwise the buffer contains the first sector of the File Allocation Table (FAT).

The device driver must perform the following actions:

- Set the pointer to the BPB table.
- Update the media descriptor.
- Set the status word in the request header.

The device driver must determine the media type in the drive in order to return the pointer to the BPB table. Previously, the FAT ID byte determined the structure and layout of the media. Because the FAT ID byte has only eight possible values (F8 through FF), it is clear that, as new media types are invented, the available values will soon be

exhausted. With the varying media layouts, OS/2 needs to be aware of the location of the FATs and directories before it reads them.

The device driver reads the boot sector from the specified buffer. If the boot sector is for DOS 2.00, 2.10, 3.10, 3.20, or OS/2, the device driver returns the BPB from the boot sector. If the boot sector is for DOS 1.00 or 1.10, the device driver reads the first sector of the FAT into the specified buffer. The FAT ID is examined and the corresponding BPB is returned. Only two formats are possible for diskettes formatted by a 1.00 or 1.10 system, 5 1/4-inch single-sided (FEH) and 5 1/4-inch double-sided (FFH.)

The information relating to the BPB for a particular media is kept in the boot sector for the media.

Boot Sector Format

Field	Length
Short JUMP (EBH) followed by a NOP (90H)	2 BYTES
OEM name and version	8 BYTES
Bytes per sector	WORD
Sectors per allocation unit (must be a power of 2)	BYTE
Reserved sectors (starting at logical sector 0)	WORD
Number of FATs	BYTE
Number of root directory entries (maximum allowed)	WORD
Number of sectors in logical image (total sectors in media, including boot sector, directories, for example.)	WORD

2H / BUILD BPB
Build BIOS Parameter Block

Field	Length
Media descriptor	BYTE
Number of sectors occupied by a single FAT	WORD
Sectors per track	WORD
Number of heads	WORD
Number of hidden sectors	WORD

The last three WORDs above help the device driver understand the media. The number of heads is useful for supporting different multiple head drives that have the same storage capacity but a different number of surfaces. The number of hidden sectors is useful for supporting drive partitioning schemes.

For drivers that support volume identification and disk change, this call should cause a new volume identification to be read off the disk. This call indicates that the disk has legally changed.

Purpose

Read from or write to a device.
Read From (4H) / Write To (8H) / Write with Verify (9H)

Format of Request Block

Field	Length
Request header	13 BYTES
Media descriptor	BYTE
Transfer address	DWORD
Byte / sector count	WORD
Starting sector number for block device	DWORD
System File Number	WORD

Remarks

On entry, the request packet will have the media descriptor set for the drive identified in the request packet header. The transfer address is a 32-bit physical address of the buffer for the data. The byte/sector count is set to the number of bytes to transfer (for character device drivers) or the number of sectors to transfer (for block device drivers). The starting sector number is set for the block device drivers. The System File Number is a unique number associated with an open request.

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.

4H, 8H, 9H / READ or WRITE
Perform I/O To A Device

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 segment swapping address for the current mode. The device driver does not need to unlock the address when the request is completed.

Note: The functions IOCtl READ and IOCtl WRITE are not supported by the new OS/2 device drivers.

Purpose

Read character from buffer but do not remove it.

Format of Request Block

Field	Length
Request header	13 BYTES
Returned character	BYTE

Remarks

The device driver must perform the following actions:

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

For input on character devices with a buffer, the device driver returns from this function with the busy bit set to 0 along with a copy of the first character in the buffer. The busy bit is set to 1 to indicate no characters in the buffer. This function allows the operating system to look ahead one input character without blocking in the device driver.

6H, AH / STATUS
Input or Output Status

Purpose

Determine input or output status on character devices.
Input Status (6H) / Output Status (AH)

Format of Request Block

Field	Length
Request header	13 BYTES

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.

For output on character devices, if the busy bit is returned set to 1, a subsequent write request to the device driver would have to wait for the completion of a currently active request. If the busy bit is returned set to 0, there is no current request. Therefore, a write request would start immediately.

For input on character devices with a buffer, if the busy bit is returned set to 1, there are no characters currently buffered in the device driver. If the busy bit is returned set to 0, there is at least one character in the device driver buffer. The effect of busy bit = 0 is that a read of one character will not need blocking. Devices that do not have an input buffer in the device driver should always return busy = 0.

Purpose

Flush or terminate all pending requests.
Input Flush (7H) / OutPut Flush (BH)

Format of Request Block

Field	Length
Request header	13 BYTES

Remarks

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the request header.

This call tells the device driver to flush (terminate) all known pending requests. Its primary use is to flush the input (or output) queue on character devices.

DH, EH / OPEN or CLOSE
Open / Close Device

Purpose

Open or close the device.
Open Device (DH) / Close Device (EH)

Format of Request Block

Field	Length
Request header	13 BYTES
System File Number	WORD

Remarks

The System File Number is a unique number associated with an open request.

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the request header.

Character device drivers may use OPEN/CLOSE requests to correlate using their devices with application activity. For instance, the device driver may increase a reference count for every OPEN and decrease the reference count for every CLOSE. When the count goes to 0, the device driver can flush its buffers. This can be thought of as a "last close causes flush," or as the device driver using the OPEN as an indicator to send an initialization string to its device.

For example, to ensure that a printer is in a known state at the start of an I/O stream, this call could be used to set the font and page size. Similarly, the CLOSE call can be used to send a post-string (like a form feed) at the end of an I/O stream. Using IOCtl to set these pre-strings and post-strings provides a flexible mechanism of serial I/O device stream control.

Purpose

Check for removable media.

Format of Request Block

Field	Length
Request header	13 BYTES

Remarks

The device driver must perform the following actions:

- Set the busy bit of the status word.

  Set the busy bit to 1 if the media is non-removable. Set the busy bit to 0 if the media is removable.

- Set the status word in the request header.

The device driver receives this request packet when an application issues an IOCtl function call to determine whether it is dealing with a removable or non-removable media drive. For example, removable or non-removable drives may print different versions of some prompts.

10H / GENERIC IOCtl
I/O Control for Devices

Purpose

Send I/O control commands to a device.

Format of Request Block

Field	Length
Request header	13 BYTES
Function category	BYTE
Function code	BYTE
Parameter Buffer Address	DWORD
Data Buffer Address	DWORD
System File Number	WORD

Remarks

On entry, the request packet will have the IOCtl category code and function code set. The parameter buffer and the data buffer addresses will be set as virtual addresses. Note that some IOCtl functions do not require data and/or parameters to be passed. For these IOCtls, the parameter and data buffer addresses may contain zeros. The System File Number is a unique number associated with an open request.

The device driver must perform the following actions:

- Perform the requested function.
- Set the status word in the request header.

The device driver is responsible for locking the parameter and data buffer segments, and converting the pointers to 32-bit physical addresses if necessary.

Refer to *Technical Reference, Vol. 2* for more detailed information.

Purpose

Reset the Uncertain Media error condition and allow OS/2 to identify the media.

Format of Request Block

Field	Length
Request header	13 BYTES

Remarks

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

The device driver must perform the following actions.

- Set the status word in the request header.
- Reset the error condition for the drive.

Previous to this command, the device driver had returned the error "Uncertain Media" for the drive. This action informs the device driver that it no longer needs to return the error for the drive.

12H, 13H / LOGICAL DRIVE
Get/Set Logical Drive Mapping

Purpose

Get/Set which logical drive is currently mapped onto a particular unit.
Get Logical Drive Mapping (12H) / Set Logical Drive Mapping (13H)

Format of Request Block

Field	Length
Request header	13 BYTES

Remarks

On entry, the unit code 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 number in the request header.
- For Set, it must map the logical drive represented by the unit number onto the physical drive that has the mapping of logical drives.
- The logical drive is returned in the unit code field. This field is set to 0 if there is only one logical drive mapped onto the physical drive.
- Set the status word in the request header.

Purpose

Terminate the character device driver.

Format of Request Block

Field	Length
Request Header	13 BYTES

Remarks

When a device driver is loaded, the attribute field and name in its header are used to determine if the new device driver is attempting to replace a driver (device) already installed. If so, the previously installed device driver is requested by the operating system to DEINSTALL the indicated device. If the installed device driver refuses the DEINSTALL command, then the new device driver is not allowed to initialize. If the installed device driver performs the DEINSTALL, then the new device driver is initialized.

If the character device driver honors the DEINSTALL request, it must perform the following actions:

- Release any allocated physical memory.
- UnSet any hardware vectors that it had claimed.
- If the device driver has a software interrupt handler, it cannot reset the vector, rather it must preserve the DOS mode vector chain by doing a JMP to the previous handler.
- Perform any other cleanup.
- Clear the error bit in the status field to indicate a successful DEINSTALL.

14H / DEINSTALL
Terminate the Device Driver

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

- Set the error bit in the status field and set the error code to 03H, UNKNOWN COMMAND.

DEINSTALL Considerations

Logical IDs: An ABIOS device driver maps its device name to a unit within a Logical ID (LID). It receives a DEINSTALL request for its device name, which implies a single unit of a LID. To honor the DEINSTALL request, it must relinquish the LID via the DevHlp FreeLIDEntry at DEINSTALL time.

Note: To release a LID means to release all units under that LID. For a LID that has multiple units, the device driver must discontinue support of all units under the LID. If multiple units correspond with multiple device headers in the device driver data segment, the device driver must note which device header corresponds to each unit in the DEINSTALL LID, and discontinue support.

Hardware Interrupts: In honoring a DEINSTALL command, a device driver must remove its claim on the interrupt level. The DevHlp UnSetIRQ provides this service.

If the device driver's device is ill-behaved (that is, it cannot be told to stop generating interrupts or be quiesced), the device driver must not remove its interrupt handler. In this case, the device driver must refuse the DEINSTALL request.

Note: Because of the general interrupt sharing capabilities in a level-sensitive interrupt environment, device drivers should not assume that the DevHlp SetIRQ service can be used to determine whether a given device is being used by another device driver. Instead, the DEINSTALL convention should be used on the logical device name that another device driver may be using to access the same device.

Purpose

This call is used by the system to ask the device driver how many physical-partitionable fixed disks the device driver supports.

Format of Request Block

Field	Length
Request Header	13 BYTES
Count	BYTE
Reserved	WORD
Reserved	WORD

Remarks

This is done to allow the Category 9 Generic IOCtls to be appropriately 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.

The device driver must perform the following actions:

- Set the count as discussed above (1-based).
- Set the status word in the request header.

17H / GET FIXED DISK/LOGICAL UNIT MAP

Purpose

This call is used by the system to determine which logical units supported by the device driver exist on physical partitionable fixed disk N.

Format of Request Block

Field	Length
Request Header	13 BYTES
Units-supported bit mask	4 BYTES
Reserved	WORD
Reserved	WORD

Remarks

On entry,the request packet header unit field identifies a physical disk number (based on 0) instead of a logical unit number. The device driver returns a bit map of which logical units exist on the physical drive. The physical drive relates to the partitionable fixed disks reported to the system by way of the PARTITIONABLE FIXED DISKS command. It is possible that no logical units exists on a given physical disk because it has not yet been initialized.

The device driver must perform the following:

- Set the 4-byte bit mask to indicate which logical units that it owns exist on the physical partitionable fixed disk for which the information is being requested.
- Set the status in the request packet header.

The bit mask is set up as follows. A 0 means the logical unit does not exist and a 1 means it does. The first logical unit that the device driver supports is the low-order bit of the first BYTE. The bits are used from right to left in the diagram below starting at the low order bit of each following BYTE. It is possible that all the bits will be 0.

For example, a block device driver supports five units spread over the two diskette drives and one partitionable fixed disk in a system. Unit 0 and unit 1 map to the diskette drives. Unit 2, 3, and 4 map to the fixed disk. For the device command, this device driver will set the 4-byte bit map to:

```
'0000 0000 0000 0000 0000 0000 0001 1100' binary
```

or

```
'00 00 00 1C' hex
```

Device Driver Examples

Using PhysToVirt and UnPhysToVirt: There are some basic guidelines when using the DevHlp services for address conversion, PhysToVirt and UnPhysToVirt.

- Use ES:DI whenever possible when converting a single physical address.
- Use ES:DI for the first address conversion when using two physical addresses.
- Check the physical address pair and convert the physical address that lies above 1 Mb first.

The following examples show the recommended way to use these DevHlps in various scenarios. These examples apply to both task-time and interrupt-time operations, except where noted.

- To get a logical address to place in an ABIOS request block:
  1. Call PhysToVirt with ES:DI for the converted address.
  2. Store the converted address in the ABIOS request block.
  3. Call the ABIOS service.
  4. Call UnPhysToVirt.
- To convert a single physical address to use as the source in a data transfer to a logical address, (that is, one that was passed as input for this data transfer request):
  1. Save DS.
  2. Call PhysToVirt with DS:SI for the converted address.
  3. Perform the data transfer.
  4. Restore DS.
  5. Call UnPhysToVirt.

- To provide two logical addresses in order to do a data transfer:
  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer.
  6. Restore DS.
  7. Call UnPhysToVirt.
- To Do multiple data transfers:
  1. Examine the first physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer.
  6. Restore DS.
  7. Examine the second physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  8. Call PhysToVirt with ES:DI for the first address.
  9. Save DS.
  10. Call PhysToVirt with DS:SI for the second address.
  11. Perform the data transfer.
  12. Restore DS.
  13. Perform these steps until all data transfers are complete.
  14. Call UnPhysToVirt.

- To provide two logical addresses for a data transfer which must be broken down into smaller chunks in order to YIELD periodically:

  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer on the chunk.
  6. Restore DS.
  7. Call UnPhysToVirt.
  8. YIELD.
  9. When control is returned, repeat these steps.

- To pass two logical addresses to a subroutine, one of which must be converted from a physical address, the other is obtained from the device driver's data segment:

  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  2. Call PhysToVirt with ES:DI for the address to convert.
  3. Save the converted address in the appropriate input parameter to the subroutine.
  4. Save the other logical address (located in the device driver's data segment) in the appropriate input parameter to the subroutine.
  5. Call the subroutine.
  6. Call UnPhysToVirt.

- To use two logical addresses to do a data transfer at interrupt time before the EOI:
  1. Examine the physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  2. Call PhysToVirt with ES:DI for the first address.
  3. Save DS.
  4. Call PhysToVirt with DS:SI for the second address.
  5. Perform the data transfer.
  6. Restore DS.
  7. Call UnPhysToVirt.
  8. Issue the EOI.
- To use two logical addresses in order to do a data transfer at interrupt time after the EOI:
  1. Issue the EOI.
  2. Examine the physical address pair. If one of the physical addresses is above 1 MB, then convert it first.
  3. Save the interrupt flag.
  4. Disable interrupts.
  5. Call PhysToVirt with ES:DI for the first address.
  6. Save DS.
  7. Call PhysToVirt with DS:SI for the second address.
  8. Perform the data transfer.
  9. Restore DS.
  10. Restore the interrupt flag.
  11. Call UnPhysToVirt.

Notes On Writing a Device Driver using Advanced BIOS

The following is a high-level example of how a device driver would be written to use Advanced BIOS.

To determine the basic structure of the device driver, certain design points must be identified.

- The kind of device to be supported (character or block).
- The nature of the I/O to the device (synchronous or staged, Program I/O (PIO) or Direct Memory Access (DMA)).

  A staged request can be further refined to be staged on a time delay, staged on an interrupt or both. Staged on a time delay means the operation involves waiting for a specific length of time before the operation can be continued or is completed. Staged on an interrupt means the operation involves waiting for an interrupt to occur.

  PIO or DMA refers to the type of addressing required for data transfers. PIO is done using virtual addresses (which are also referred to as logical addresses) of the form:

  ```
  segment/selector:offset.
  ```

  DMA is done using physical addresses which are 32-bit numbers indicating the data transfer location in memory.

- The maximum number of devices.
- The maximum number of interrupt levels.

These items determine the nature of the device driver, that is, how the task-time and interrupt-time portions of the device driver relate to each other and which of the DevHlp services will be used for blocking, queueing, timers, and others.

Note that the type and number of devices generally indicate the logical device names (for example, COM1, LPT1) that the device driver will support.

- A device type is identified by its ABIOS-architected device ID.
- A specific device is identified by a Logical ID (LID) and unit number under that LID.
- I/O to the device is performed by calling the ABIOS entry point (Start, Interrupt, or Timeout) that corresponds to the particular LID.
- Parameters are passed to an ABIOS service through a request block structure.
- I/O requests can be synchronous (run to completion), or staged (run until blocked, waiting for an interrupt or time).
- Staged requests may have well-defined time delays between certain stages.
- Data transfers may use either virtual addresses

  `(segment/selector:offset)`

  or physical addresses.

Before using ABIOS services and during initialization, a device driver must identify every LID for which it will accept requests. To do this, the device driver uses the architected ABIOS Device ID for its device. The device driver uses the DevHlp GetLIDEntry, which searches through the Advanced BIOS common data area looking for the LID that corresponds to the given device ID.

In general, by making repeated calls to GetLIDEntry and counting the number of units supported by each LID it obtains, the device driver determines how many supported devices are configured in the system. The device driver will only process interrupts and requests for its maximum number of supported devices. Any LID of the device driver's device type that is leftover must be unclaimed so another device driver can support it.

A device driver knows which LID corresponds to a given logical device name (for example, COM1) because of the rule forcing the operating system logical device names to be in the same order as the LID entries for an associated device ID.

For example, assuming one unit per LID, then an installable printer device driver will support LPT3 (the third printer) by locating the third LID that corresponds to device ID of printer (that is "awake").

The device driver must determine which interrupt level each LID will use by using the ABIOS function, Return LID Parameters. The device driver will register interrupt handler entry points for the interrupt levels that it supports with the DevHlp SetIRQ. It keeps a list of every LID that corresponds to each interrupt handler.

Note: If the device driver supports multiple devices and the number of interrupt levels for those devices exceed the number of supported interrupt levels, the device driver will ignore any LID that it cannot support because too many different interrupt levels are required.

At task time, when the device driver strategy entry point for a given device header receives a request packet, the device driver knows which logical device name and LID (and unit number) correspond to that entry point.

The device driver strategy routine sets up an ABIOS request block and uses the DevHlp ABIOSCall to invoke the Advanced BIOS START routine to begin the requested ABIOS function. ABIOS requires that the ReturnCode field in the ABIOS request block be initialized to FFFFH. ABIOS will set the ReturnCode to its appropriate value.

Note: Either portion of the device driver, the task-time strategy routine or the interrupt handler, may start an ABIOS request. For simplicity, the example will use the strategy routine as the caller of the Advanced BIOS START service.

The pointer to the ABIOS request block and any logical data transfer pointers can be set up by the device driver independent of mode if the data transfer areas are in the device driver's data segment.

```
(segment/selector:offset)
```

If the data transfer is to take place to a logically addressed buffer in a bimodal environment, the device driver will need to double buffer the data transfer if the target location of the data could be high memory while running in real mode or if the device driver needs to run enabled. That is, the device driver should use an intermediate buffer located in its data segment (which is below 1 MB) for the data transfer performed by ABIOS. The device driver would then complete the data transfer to the user data buffer itself.

In a multi-staged request, the address of the logically addressed data buffer may have to be changed by the device driver in the Advanced BIOS request block from stage to stage. This is normal because of the bimodal characteristics of the operating environment. For devices that require physical address data transfer (for example, DMA-oriented devices), the device driver must ensure that the buffer area is locked for the duration of all stages of the request.

Interrupt during START: If the request is staged on an interrupt then ABIOS will set the ReturnCode appropriately only when the particular service is ready to be resumed through the Advanced BIOS INTERRUPT routine. The device driver strategy routine must also set a flag to indicate whether a request has completed the START request to the point at which the strategy routine interrogates the ReturnCode. This must be done to accommodate the case where the interrupt occurs after ABIOS updates the ReturnCode, but before the device driver strategy routine interrogates the ReturnCode. In this case, the device driver Interrupt handler is invoked by the interrupt and can take appropriate action on the request block, even though the device driver strategy routine has not completed processing the request block.

For example, if the strategy routine is expected to BLOCK the request until the interrupt occurs, but the interrupt handler is invoked before the strategy routine is able to BLOCK, the interrupt handler needs to flag the fact that the interrupt handler already processed the request block. The strategy routine, when it gets control, will then see that it should not check the ReturnCode in the request block and it does not have to BLOCK because the request is already completed. The strategy routine will set up the request packet with the return information and return the completed request to the kernel.

This example would be more complex if, when the strategy routine got control, the request was still incomplete (as would occur in a multi-staged request). The strategy routine would still ignore the ReturnCode because the request block would already be at a different stage (than the START) but the strategy routine may still have to BLOCK.

Interrupt after START: For a staged ABIOS request that must wait for the interrupt associated with the specified LID to occur after the request is STARTed, the ReturnCode of the request will be set to stage-on interrupt by the ABIOS START function. This indicates that the request is incomplete. Several requests for this LID may START and be waiting for the device interrupt. These incomplete requests are commonly referred to as outstanding requests for the LID.

Note: The request is considered to be an outstanding request for the LID, even if the START service has not returned control to the caller of the START service.

A device driver never assumes that the return codes for an ABIOS request occur in any given order. The ReturnCode should always be checked to determine what actions to perform on the request block.

When the device driver interrupt handler is invoked by the device interrupt, it knows which LID is associated with the interrupt level. The interrupt handler must individually examine each LID associated with the interrupt level. For a LID, the interrupt handler must process all outstanding staged-on interrupt request blocks. That is, the Interrupt handler is required by ABIOS to call the Advanced BIOS INTERRUPT routine for every outstanding staged-on interrupt request block to completely process one LID. This includes a START request block in which the ReturnCode has been changed from FFFFH to stage-on interrupt but where the START service has not yet returned control to its caller. If one of the request blocks for the LID caused the interrupt, then after the interrupt handler has called ABIOS with all the outstanding request blocks owned by this LID, the interrupt handler will not need to process any other LID associated with this interrupt level.

If a given LID has no outstanding ABIOS request blocks, the device driver will call the Advanced BIOS DEFAULT INTERRUPT service for that LID. The DEFAULT INTERRUPT service will reset the interrupt condition for that LID if the LID falsely caused the interrupt. It will then return to the device driver interrupt handler, indicating that the interrupt belonged to the LID.

If there is at least one outstanding ABIOS request block for a given LID, ABIOS will automatically invoke the DEFAULT INTERRUPT service if the LID generates a false interrupt. The device driver must be able to process the false interrupt return code for any call to the ABIOS INTERRUPT routine. This return code indicates that the interrupt belonged to the LID, was reset by ABIOS, and the device driver is responsible for issuing the EOI and returning to the operating system as owning the interrupt. The device driver is still responsible for calling ABIOS with any remaining staged-on-interrupt request blocks for this LID.

The device driver interrupt handler may issue the EOI (via the DevHlp EOI function) only after completely processing the LID that owns the interrupt or after the LID's DEFAULT INTERRUPT service indicates that the LID's device caused the interrupt. In other words, the Interrupt handler must process all outstanding requests under the LID that owns the interrupt, even after finding a request block which indicates that it caused the interrupt. The interrupt handler may stop processing any LID for this interrupt only when the interrupt is claimed by a LID, either by a request under the owning LID or by the owning LID's DEFAULT INTERRUPT service.

The interrupt handler knows that once it issues the EOI after completely processing a LID, another LID requesting service at the current interrupt level, would create another interrupt.

Once the EOI is issued, the interrupt handler can be reentered at its entry point. If the Interrupt handler is reentered, it must process every LID, including the one that is near completion or just completed.

In order to keep the pre-EOI processing time to a minimum, the interrupt handler may wish to issue its EOI either before it sets the return information in the operating system request packet, or before it begins processing a request packet that was queued by the device driver strategy routine.

If the interrupt handler did not find any LID that claimed the given interrupt either with a request block or by a DEFAULT INTERRUPT service, the interrupt handler must exit, indicating that the interrupt did not belong to it; that is, it was not caused by any LID that the device driver owns.

Eventually, the ReturnCode from ABIOS will show that the ABIOS request block is complete. The device driver can then clean up the device driver request packet queue and take the next request packet and try to start another ABIOS request block.

The device driver will support the timeout requirements of Advanced BIOS with the DevHlp SetTimer and the DevHlp TickCount. Instead of counting every clock tick, the device driver will use TickCount to force its timer tick entry point to receive control as infrequently as possible. The design of the Advanced BIOS TIMEOUT function is in 1-second increments.

Device drivers in an Advanced BIOS environment have the same requirement to support DOS mode operations as they do in OS/2. In addition, for certain devices such as diskette and disk, the device driver must reset the device when switching between Advanced BIOS operation and BIOS operation. This is because state of the device information is kept internally to the Advanced BIOS and BIOS device blocks, which would get out of synchronization if the device were not reset when switching between the two sets of code.

Chapter 8. Device Helper Services

Many of the functions of an OS/2 device driver are related to system operations rather than to hardware operations. Therefore, 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 pointer to the DevHlp interface. The pointer to the DevHlp interface is a bimodal pointer; that is, this pointer to the DevHlp interface is valid in both the DOS mode and the OS/2 mode. The device driver does not have to be sensitive to the mode of operation before requesting DevHlp services.

The DevHlp services are listed below. 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 driver initialization time.

DevHlp Services and Function Codes

DevHlp Service	Code	Description
SchedClockAddr	0	Get system clock routine
DevDone	1	Device I/O complete
Yield	2	Yield the CPU
TCYield	3	Yield the CPU to time-critical
Block	4	Block thread on event
Run	5	Unblock thread
SemRequest	6	Claim a semaphore
SemClear	7	Release a semaphore
SemHandle	8	Get a semaphore handle
PushReqPacket	9	Add request to list
PullReqPacket	A	Remove request from list
PullParticular	B	Remove a specific request from list
SortReqPacket	C	Insert request in sorted order to list

DevHlp Service	Code	Description
AllocReqPacket	D	Get a request packet
FreeReqPacket	E	Free request packet
QueueInit	F	Initialize character queue
QueueFlush	10	Clear character queue
QueueWrite	11	Put a character in the queue
QueueRead	12	Get a character from the queue
Lock	13	Lock segment
Unlock	14	Unlock segment
PhysToVirt	15	Map physical-to-virtual address
VirtToPhys	16	Map virtual-to-physical address
PhysToUVirt	17	Map physical-to-user virtual
AllocPhys	18	Allocate physical memory
FreePhys	19	Free physical memory
SetROMVector	1A	Set Software interrupt vector
SetIRQ	1B	Set a hardware interrupt handler
UnSetIRQ	1C	Reset a hardware interrupt handler
SetTimer	1D	Set a timer handler
ResetTimer	1E	Remove a timer handler
MonitorCreate	1F	Create a monitor
Register	20	Install a monitor
DeRegister	21	Remove a monitor
MonWrite	22	Pass data records to monitor
MonFlush	23	Remove all data from stream
GetDOSVar	24	Return pointer to DOS variable
SendEvent	25	Indicate an event
ROMCritSection	26	ROM BIOS critical section
VerifyAccess	27	Verify memory access
Reserved	28	
Reserved	29	
Reserved	2A	
Reserved	2B	
Reserved	2C	
AllocGDTSelector	2D	Allocate GDT Selectors
PhysToGDTSelector	2E	Map physical to virtual address
RealToProt	2F	Real Mode to Protect Mode
ProtToReal	30	Protect Mode to Real Mode
EOI	31	Issue an End-Of-Interrupt

DevHlp Service	Code	Description
UnPhysToVirt	32	Mark PhysToVirt complete
TickCount	33	Modify timer
GetLIDEntry	34	Get Logical ID
FreeLIDEntry	35	Release Logical ID
ABIOSCall	36	Invoke ABIOS function
ABIOSCommonEntry	37	Invoke ABIOS Common Entry Point
Reserved	38	

As discussed in the section on 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 DOS mode software interrupt runs.
- Init mode - the context in which the device driver Strategy routine runs when it is called with the INIT request packet.

DevHlp Services and Corresponding States

Certain restrictions apply as to when individual DevHlp services can be used. The following list outlines which DevHlp services are allowed in which contexts (kernel, interrupt, user, or initialization).

DevHlp Service	Code	Kernel	Interrupt	User	Init
SchedClockAddr	0	*			*
DevDone	1	*	*		
Yield	2	*			
TCYield	3	*			
Block	4	*		*	

DevHlp Service	Code	Kernel	Interrupt	User	Init
Run	5	*	*	*	
SemRequest	6	*		*	
SemClear	7	*	*	*	
SemHandle	8	*	*		
PushReqPacket	9	*			
PullReqPacket	A	*	*		
PullParticular	B	*	*		
SortReqPacket	C	*			
AllocReqPacket	D	*			
FreeReqPacket	E	*			
QueueInit	F	*	*	*	*
QueueFlush	10	*	*	*	
QueueWrite	11	*	*	*	
QueueRead	12	*	*	*	
Lock	13	*			
Unlock	14	*			*
PhysToVirt	15	*	*		*
VirtToPhys	16	*			*
PhysToUVirt	17	*			
AllocPhys	18	*			*
FreePhys	19	*			*
SetROMVector	1A	*			*
SetIRQ	1B	*			*
UnSetIRQ	1C	*	*		*

DevHlp Service	Code	Kernel	Interrupt	User	Init
SetTimer	1D	*			*
ResetTimer	1E	*	*		*
MonitorCreate	1F	*			*
Register	20	*			
DeRegister	21	*			
MonWrite	22	*	*	*	
MonFlush	23	*			
GetDOSVar	24	*			*
SendEvent	25	*	*		
ROMCritSection	26			*	
VerifyAccess	27	*			
AllocGDTSelector	2D				*
PhysToGDTSelector	2E	*	*	*	*
RealToProt	2F		*		
ProtToReal	30		*		
EOI	31		*		*
UnPhysToVirt	32	*	*		*
TickCount	33	*	*	*	*
GetLIDEntry	34	*			*
FreeLIDEntry	35	*			*
ABIOSCall	36	*	*	*	*
ABIOSCommonEntry	37	*	*	*	*

Related DevHlp services are grouped together in the following categories.

1. System Clock Management
    - SchedClockAddr
2. Process Management
    - Block
    - DevDone
    - Run
    - TCYield
    - Yield
3. Semaphore Management
    - SemClear
    - SemHandle
    - SemRequest
4. Request Queue Management
    - AllocReqPacket
    - FreeReqPacket
    - PullParticular
    - PullReqPacket
    - PushReqPacket
    - SortReqPacket

5. Character Queue Management

    - QueueFlush
    - QueueInit
    - QueueRead
    - QueueWrite

6. Memory Management

    - AllocGDTSelector
    - AllocPhys
    - FreePhys
    - Lock
    - PhysToGDTSelector
    - PhysToUVirt
    - PhysToVirt
    - Unlock
    - UnPhysToVirt
    - VerifyAccess
    - VirtToPhys

7. Interrupt Management

    - EOI
    - SetIRQ
    - SetROMVector
    - UnSetIRQ

8. Timer Services

    - ResetTimer
    - SetTimer
    - TickCount

9. Monitor Management

    - DeRegister
    - MonFlush
    - MonitorCreate
    - MonWrite
    - Register

10. System Services

    - GetDOSVar
    - SendEvent
    - ROMCritSection

11. Processor Mode Services

    - ProtToReal
    - RealToProt

12. Advanced BIOS Services

    - ABIOSCall
    - ABIOSCommonEntry
    - FreeLIDEntry
    - GetLIDEntry

DevHlp Interfaces

Calling conventions for each of the DevHlp services follow. In addition to the explicit effects noted under each service, the interrupt flag can be set or cleared by some services, and other flags can be affected by the calls. Some services require that the interrupt flag be off when they are called.

The device driver can assume that the state of the interrupt flag will be preserved, and that the DevHlp routine will not enable interrupts unless stated otherwise in the functional description for each routine. The only exceptions apply to functions that allow the device driver to relinquish control of the CPU. Therefore, during calls to functions such as Yield and TCYield, the device driver cannot assume that interrupts will remain disabled.

All registers except flag registers are preserved across DevHlp calls unless specified as containing return parameters.

The functional descriptions for the DevHlp services follow in alphabetical order.

ABIOSCall
Invoke ABIOS function

Purpose

This routine is used to invoke an ABIOS service for the Operating System Transfer Convention.

Processing

```
MOV  AX,LID              ;Logical ID
MOV  SI,RB_Offset        ;Offset in data segment
                         ; to ABIOS request block
MOV  DH,Entry_Point      ;Specifies entry point
                         ; 0 = start
                         ; 1 = interrupt
                         ; 2 = timeout
MOV  DL,DevHlp_ABIOSCall
CALL [Device_Help]
```

Results

```
'C' Clear if no error
    ABIOS service invoked.
'C' Set if error
    AX = error code
         ABIOS not present.
       Unknown ABIOS Command
```

Remarks

The stack is set, depending on the current address mode, for the call to ABIOS, and the indicated ABIOS function is called according to the Operating System Transfer Convention. Refer to Operating System Transfer Convention on page 7-33. When the ABIOS function returns, ABIOSCall will clean up the stack before returning to the device driver.

Note: Advanced BIOS functions called in user mode may need to be protected from being suspended in the background. This will occur when the DOS mode is in the foreground and the user selects an OS/2 mode application to run. The DOS mode will be suspended. To protect the ABIOS function, the device driver should issue the DevHlp call ROMCritSection.

Note that DS must point to the device driver's data segment. If DS had been previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

ABIOSCommonEntry
Invoke ABIOS Common Entry Point

Purpose

This service is used to invoke an ABIOS Common Entry Point according to the Advanced BIOS Transfer Convention.

Processing

```
MOV  SI,RB_Offset          ;Offset in data segment
                           ; to ABIOS request block
MOV  DH,Entry_Point        ;Specifies entry point
                           ; 0 = start
                           ; 1 = interrupt
                           ; 2 = timeout
MOV  DL,DevHlp_ABIOSCommonEntry
CALL [Device_Help]
```

Results

```
'C' Clear if no error
    ABIOS service invoked.
'C' Set if error
    AX = error code
         ABIOS not present.
         Unknown ABIOS Command
```

Remarks

ABIOSCommonEntry sets up the stack, depending on the current address mode, for the call to one of the Advanced BIOS Common Entry Points. It then invokes the indicated ABIOS common entry point. On return from the ABIOS function, the ABIOSCommonEntry cleans up the stack before returning to the device driver.

Note: Advanced BIOS functions called in user mode may need to be protected from being suspended in the background. This will occur when the DOS mode is foreground and the user selects an OS/2 mode application to run. The DOS mode will be suspended. To protect the ABIOS function, the device driver should issue the DevHlp call ROMCritSection.

Note that DS must point to the device driver's data segment. If DS had been previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

AllocGDTSelector
Allocate GDT Selectors

Purpose

This function allocates a set of GDT selectors for a device driver to use. This allocation is performed at device driver INIT time.

Processing

```
MOV  ES,address_high          ; 32-bit address of GDT
MOV  DI,address_low           ;   selector array
MOV  CX,number                ;Number of selectors requested
MOV  DL,DevHlp_AllocGDTSelector
CALL [Device_Help]
```

Results

```
'C' Cleared if successful
'C' Set if error
    AX = error code
         Invalid address
         Zero selectors requested
         Not enough selectors available
```

Remarks

AllocGDTSelector is used to allocate a set of GDT selectors for a device driver to use for bimodal task-time and interrupt-time operations.

The address passed in ES:DI gives the location of an array of words to be filled in with the GDT selectors allocated. The value of CX specifies the number of selectors to be allocated. Note that the selector values returned may not be contiguous values.

A bimodal device driver supports both real mode and protect mode I/O and must be able to transfer data without being dependent on the current mode of operations. In addition, the interrupt handler of a bimodal device driver must be able to address data buffers regardless of the context of the current process (the current LDT will not necessarily address the data space that contains the data buffer that

the interrupt handler needs to access). The PhysToGDTSelector function is used to establish the addressability of a GDT selector, and the GDT selector's addressability will remain valid and the same until another PhysToGDTSelector call is made for the same selector.

The RealToProt function is used to change processor mode from real mode to protect mode, and the ProtToReal function is used to return the processor to real mode. The device driver must always return the processor to the same mode it was in when entered if either the ProtToReal or RealToProt functions are used.

AllocPhys
Allocate Fixed Block of Physical Memory

Purpose

AllocPhys is used by device drivers to allocate a block of fixed memory.

Processing

```
MOV  BX,size_low              ;size in bytes
MOV  AX,size_high
MOV  DH,high_or_low           ;Relative position to 1 megabyte
                              ; = 0  above 1 megabyte
                              ; = 1  below 1 megabyte
MOV  DL,DevHlp_AllocPhys
CALL [Device_Help]
```

Results

```
'C' Clear if memory allocated
    AX:BX = 32-bit physical address.
'C' Set if memory not allocated
    AX = error, memory not allocated.
```

Remarks

The memory allocated by this function is fixed memory, and may not be "unfixed" via the Unlock call.

If memory is requested to be allocated high (above 1 megabyte), and no memory above 1 megabyte is available, then an error is returned. The device driver could then attempt to allocate low memory.

Conversely, if memory is requested to be allocated low (below 1 megabyte), and no memory below 1 megabyte is available, then an error is returned and the device driver could try allocating high memory, if appropriate.

Purpose

This service returns a pointer to an empty request packet.

Processing

```
MOV  DH,wait_flag                ;Wait for available
                                 ; request packet
                                 ; = 0 if to wait
                                 ; = 1 if not to wait
MOV  DL,DevHlp_AllocReqPacket
CALL [Device_Help]
```

Results

```
'C' Cleared if a request packet was allocated.
    ES:BX  is the virtual address of
      the allocated request packet.
'C' Set if a request packet was not allocated.
```

Remarks

AllocReqPacket returns a bimodal pointer to a maximum size request packet. The bimodal pointer is a virtual address that is valid for both the DOS mode and the OS/2 mode.

Some device drivers, notably the disk device driver, need to have additional request packets to service task-time requests. Because 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 may be placed in the request packet linked list.

AllocReqPacket
Get a Request Packet

Request packets allocated in this manner should be returned to the kernel as soon as possible via the FreeReqPacket function. The system as a whole 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.

The state of the interrupt flag is not preserved across calls to this DevHlp.

Purpose

This service "sleeps" the thread executing in the device driver until either the Run call is issued on the event identifier or a timeout occurs.

Processing

```
MOV  BX,event_id_low          ;Low word of event id
MOV  AX,event_id_high         ;High word of event id
MOV  DI,time_limit_high       ;Timeout interval
MOV  CX,time_limit_low        ;in milliseconds
                              ; = -1 if to never timeout
MOV  DH,interruptible_flag    ;Tells if sleep is interruptible
                              ; = 0  interruptible
                              ; = 1  non-interruptible
MOV  DL,DevHlp_Block
CALL [Device_Help]
```

Results

```
'C' Clear if event wakeup.
'C' Set if unusual wakeup.
'Z' Set if wakeup due to timeout.
'Z' Cleared if sleep was interrupted.
AL  Awake code, non-zero if unusual wakeup.
    Interrupts enabled.
```

Remarks

The return from the Block call indicates whether the "wakeup" occurred from a normal of Run call or an expiration of the time limit.

Block removes the current thread from the run queue and starts executing some other thread. The thread blocked in the device driver is reactivated and Block returns when Run is called with the same event identifier, when the time limit expires, or when the thread is signalled.

Block
This Thread From Running

The event identifier is an arbitrary 32-bit value, but a convention must be followed 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 blocking until some resource is cleared normally blocks "on" the address of the ownership flag for that resource.

Because dual-mode device drivers may be Blocked in one mode and Run in the other, using the virtual address as the event identifier is not sufficient; the virtual address of an item in one mode is not the same as its virtual address in the other mode. A possible option for a device driver is that it Blocks/Runs on the 32-bit address of a request packet being processed for the thread.

The Block/Run mechanism is so designed that it cannot guarantee immunity from a faulty wakeup by some other thread in the system running a 32-bit key value, which happens to match some unrelated blocking thread's key. The goal is to choose keys which can be known to the Blocker and the Runner and have a high likelihood of being unique. 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.

When calling Block it is important to use the sequence:

```
Disable Interrupts
while (need to wait)
      Block (value)
      Disable Interrupts
```

Interrupts are turned off *before* checking the condition (I/O done, resource freed, whatever) first to avoid a deadlock by getting an interrupt-time Run call before completing the call to Block. Block reenables the interrupts. Also note the "while" clause, it is essential to recheck the awaited condition and, if necessary, 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 or not the sleep may be interrupted. If the sleep is interruptible, then the kernel 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 short, that is, less than a second.

If the return from the Block indicates that the sleep was interrupted, that means that some internal event occurred that requires attention (like a signal, process death, 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, setting the done bit, and returning the request packet to the kernel.

DeRegister
Remove Monitor

Purpose

DeRegister removes all of the monitors associated with the specified task from the specified monitor chain.

Processing

```
MOV  BX,monitor_PID          ; Process ID of monitor task
MOV  AX,monitor_handle       ; MonitorCreate handle for chain
MOV  DL,DevHlp_DeRegister
CALL [Device_Help]
```

Results

```
'C' Clear if OK
    AX = # of monitors still registered in chain,
         after Deregistration.
'C' Set if error
    AX = error code
         Invalid monitor handle
```

Remarks

This function may only be called at task time in the OS/2 mode.

To remove a monitor from a monitor chain, the device driver must supply the PID of the process that owns the monitor being removed and the handle of the monitor chain that is affected. All monitors belonging to the specified PID are removed from the monitor chain.

A single process may register more than one monitor with the same monitor chain. Therefore, DeRegister removes all monitors associated with the specified process from the specified monitor chain.

Purpose

This function signifies that a request has completed and unblocks any threads waiting in the kernel for the request.

Processing

```
LES  BX,request_packet        ;Pointer to I/O request
                              ; packet.
MOV  DL,DevHlp_DevDone
CALL [Device_Help]
```

Results

None

Remarks

This service will set the "done" bit in the status field of the request packet header and issue RUNs on thread(s) which are blocked in the kernel waiting for the request packet to be completed. DevDone is called from the device interrupt routine. The device driver should set any error flags in the status field before calling the routine.

Because the virtual address of a request packet is bimodal (the virtual address is valid in both the DOS mode and the OS/2 mode), the device driver may pass the request packet pointer to DevDone without being sensitive to the mode of operations.

DevDone does not apply to request packets that were allocated from the AllocReqPacket function call.

The device driver does not have to call DevDone for requests that are completed at task time (in the strategy routine). Requests that are completed at strategy time should return with the done bit set in the request packet.

EOI
Issue an End-Of-Interrupt

Purpose

This routine is used to issue an End-Of-Interrupt to the master/slave 8259 interrupt controller(s) as appropriate to the interrupt level.

Processing

```
MOV  AL,IRQnum              ;Interrupt level number
                            ; ( 0 - F )
MOV  DL,DevHlp_EOI
CALL [Device_Help]
```

Results

None

Remarks

This routine is used to issue an End-Of-Interrupt to the 8259 interrupt controller(s) on behalf of a device driver interrupt handler. If the specified interrupt level is for the slave 8259 interrupt controller, then this routine will issue the EOI to both the master and slave 8259s.

Device drivers must use this service in their interrupt handlers for upward compatibility.

Note that this routine is callable at Init-time for *interrupt processing*.

This DevHlp does not change the state of the interrupt flag. If the device driver returns to the operating system immediately after issuing the EOI, then it should disable interrupts prior to the EOI. Disabling interrupts prior to issuing the EOI allows the processing for this interrupt level to be completed before the system services the next interrupt. This reduces the probability of excessive nested interrupts causing a system stack overflow.

If any post EOI work is done by the INT handler, it should be limited to the first or non-nested interrupt. Nested INT processing should be done only prior to the EOI.

FreeLIDEntry
Release a Logical ID

Purpose

This routine is used to release a Logical ID. This must be done at DEINSTALL or termination time.

Processing

```
MOV  AX,LID                    ;Logical ID from
                               ; GetLIDEntry
MOV  DL,DevHlp_FreeLIDEntry
CALL [Device_Help]
```

Results

```
'C' Clear if no error
'C' Set if error
    AX = Error Code
         Not your LID.
         LID does not exist.
         ABIOS not present.
```

Remarks

The attempt to free a Logical ID may fail if the device driver does not own the LID or if the LID does not exist.

Note that DS must point to the device driver's data segment. If DS was previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

Purpose

FreePhys is used to release memory allocated by the AllocPhys call.

Processing

```
MOV  BX,address_low          ;32-bit physical address
MOV  AX,address_high         ;
MOV  DL,DevHlp_FreePhys
CALL [Device_Help]
```

Results

```
'C' Cleared if memory freed
'C' Set if memory not freed
    Cannot free memory not allocated with AllocPhys.
```

Remarks

Any memory that the device driver allocated by way of the AllocPhys should be released prior to device driver termination.

FreeReqPacket
Free an Allocated Request Packet

Purpose

This service is used to release a request packet previously allocated by AllocReqPacket

Processing

```
LES  BX,request_packet       ;Pointer to request packet
                             ; previously allocated.
MOV  DL,DevHlp_FreeReqPacket
CALL [Device_Help]
```

Results

None

Remarks

The device driver should only free a request packet that had been previously allocated by AllocReqPacket. The DevDone service should not be used to return an allocated request packet.

The system as a whole 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.

The state of the interrupt flag is not preserved across calls to this DevHlp.

Get Address of Important DOS Variables

Purpose

Returns the address of a kernel variable.

Processing

```
MOV  AL,varnumber            ;Variable wanted.
MOV  DL,DevHlp_GetDOSVar
CALL [Device_Help]
```

Results

```
'C' Cleared if no error
    AX:BX points to the variable.
'C' Set if error
```

Remarks

The following is the list of *read only* variables:

Index	***Description of variable***
1	SysINFOseg:WORD Bimodal segment address of the System Global InfoSeg. 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	Reserved
4	VectorSDF:DWORD Pointer to the stand-alone dump facility. Valid at both task time and interrupt time.
5	VectorReboot:DWORD Pointer to restart the OS/2. Valid at both task time and interrupt time.

6 Reserved

Pointers to the MSATS facility, OS/2 mode and DOS mode. Valid at both task time and interrupt time.

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.

9 Reserved

10 Reserved

11 DOS mode Code Page Tag Pointer: DWORD Segment/offset of the current code page tag of DOS mode. Valid only at task time.

These variables are maintained by the kernel for the benefit of device drivers.

The returned pointer is a bimodal pointer, that is, the returned address is valid in either the DOS mode or the OS/2 mode. Note that the address returned is the "address" of the indicated variable; the variable may contain a vector to some facility or it may contain some structure.

Purpose

This routine is used to obtain a Logical ID (LID) for devices that exist (that is, devices that are "awake").

Processing

```
MOV  AL,DeviceID              ;Desired Device ID
MOV  BL,RelativeLID#          ;Nth Logical ID  of this Device ID
                              ; (0 - FF) where 0 = first unclaimed
                              ; LID (i.e., first one available)
                              ; 1 = the first LID
MOV  DH,DeviceState           ;Requested LID indicator
                              ; 0 = all other LIDs
                              ; 1 = DMA, POS
MOV  DL,DevHlp_GetLIDEntry
CALL [Device_Help]
```

Results

```
'C' Clear if no error
    AX = LID number
'C' Set if error
    AX = error code
         LID already owned.
         LID does not exist.
         ABIOS not present.
```

Remarks

GetLIDEntry is used by a device driver to obtain a LID entry. Because OS/2 does not support the Advanced BIOS Sleep/Wake functions, only devices that are "awake" are considered to exist and thus available to device drivers.

This function may be employed in two ways. One way is for the device driver to specify a relative LID. Because the ordering of LIDs corresponds to ordering of the physical devices, a device driver that desires to support a certain relative device can determine if a LID entry is available. (An example is a character device driver that sup-

ports COM4; that is, it wishes to get the LID entry for the fourth COM port.)

The other way to use this function is for the device driver to request the first available LID for its device type. (An example is a block device driver that wishes to get the first available LID for diskettes.)

In either use of this function, GetLIDEntry will search the ABIOS Common Data Area table for an entry corresponding to the specified Device ID. If an entry is located that matches the caller's form of request, it is returned to the caller. If a LID entry is found but already owned, an error is returned. If no LID entry is found, an error is also returned.

Some LIDs will not be allocated to device drivers. Certain Device IDs are used by the operating system kernel to perform such actions as mode switching. The reserved Device IDs are:

- System Services

Certain LIDs will be allocated as shared. For these Device IDs, GetLIDEntry will allow multiple device drivers to access the LID concurrently. It is up to the device driver to determine if the device is busy or available for use when needed. The list of Device IDs that are allocated as shared follows:

- DMA
- POS

Note: GetLIDEntry must be called with the DeviceState parameter set to 1 in order to obtain a LID for these Device IDs. In all other cases, DeviceState must be set to 0.

Note: DS must point to the device driver's data segment. If DS was previously used in a PhysToVirt call, it must be reset to the device driver's data segment.

Purpose

Lock is called by device drivers at task time to lock a memory segment.

Processing

```
MOV  AX,segment_@          ;Selector or segment
MOV  BH,duration           ;Advisory duration of lock
                           ; = 0 if short-term
                           ; = 1 if long-term
MOV  BL,waitflag           ;= 0 Block till locked
                           ;= 1 Return if it is not
                           ;    immediately available
MOV  DL,DevHlp_Lock
CALL [Device_Help]
```

Results

```
'C' Clear if segment locked
    AX:BX  = lock handle
'C' Set if segment unavailable or invalid handle.
    AX, BX not preserved
```

Remarks

If the segment is unavailable, the caller must specify whether Lock should block until the segment is available and locked, or return immediately.

If the advisory lock duration parameter indicates that the segment is expected to be locked (fixed) for a long time, the segment may be moved to the region reserved for fixed, OS/2 mode segments.

The duration of the lock must be set to SHORT-TERM for operations that are expected to complete in two seconds or less. Use of short-term locks for longer periods of time prevents an adequate amount of movable, swappable memory from being available for system use.

Lock
Memory Segment

Prior to requesting a lock on a process-supplied address, a device driver must verify the process' access to the memory with the DevHlp VerifyAccess. The device must not yield the CPU between the VerifyAccess and the Lock, otherwise the process could shrink the segment before it has been locked. Once the user access has been verified, the device driver may lock the segment (and convert the address to a 32-bit physical address). The access verification is valid for the duration of the lock.

The Lock call need only be done on addresses that are received from user processes (such as an address that is passed via an IOCtl).

Purpose

MonFlush removes all data from the specified monitor chain (such as the data stream).

Processing

```
MOV  AX,monitor_handle        ;MonitorCreate handle for chain
MOV  DL,DevHlp_MonFlush
CALL [Device_Help]
```

Results

```
'C' Clear if OK
    AX = 0
'C' Set if error
    AX = error code
         Invalid monitor handle
```

Remarks

This function may be called at task time in the OS/2 mode only.

The general format of monitor records requires that every record contain a flag word as the first entry. One of the flags is used to indicate that this record is a flush record. The flush record consists of only the flag word. This record is used by monitors along the chain to reset internal state information and to assure that all internal buffers are flushed. The flush record must be passed along to the next monitor because the monitor dispatcher will not process any more information until the flush record is received at the end of the chain.

Subsequent MonWrite requests will fail (or block) until the flush completes.

The state of the interrupt flag is not preserved across calls to this DevHlp.

MonitorCreate
Create a monitor

Purpose

MonitorCreate creates an initially empty chain of monitors or removes an empty chain of monitors.

Processing

```
LES  SI,final_buffer     ;Address of device driver's
                                    ; monitor chain buffer
LDS  DI,notify_rtn       ;Address of notification routine
MOV  AX,Handle           ;Handle for this chain
                         ;  = 0  create new monitor chain
                         ; < > 0  specifies chain to be removed
                                  (returned from previous create call)
MOV  DL,DevHlp_MonitorCreate
CALL [Device_Help]
```

Results

```
'C' Clear if OK
    AX = monitor chain handle if Handle was "0".
    AX = 0 if handle was not "0".
'C' Set if error
    AX = error code
         Invalid monitor handle
         Not enough memory
         Monitor chain not empty
         Invalid parameter
          (If handle not 0, this error is returned if
          all monitor tasks registered with this chain
          have been previously deregistered.)
```

Remarks

The monitor chain buffer is a buffer owned by the device driver. On calling MonitorCreate, the first word of this buffer is the length of the buffer in bytes (including the first word).

When the monitor chain handle specified is 0, a new monitor chain is created. When the monitor chain handle specified is not 0, the

monitor chain created by a previous MonitorCreate call is removed, or destroyed.

A monitor chain is a list of monitors, with a device driver monitor chain buffer address and code address as the last element on this list. Data is placed into a monitor chain via the MonWrite function; the monitor dispatcher feeds the data through all registered monitors, putting the resulting data, if any, into the specified device driver monitor chain buffer. When data is 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.

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.

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 device driver's buffer.

This routine can also destroy a monitor chain if the handle parameter (AX) is non-zero. The non-zero value is the handle of the chain to remove.

This function may only be called at task time in the OS/2 mode.

A MonitorCreate call must be made before a monitor can be Registered with the chain. This can be done at any time including during the installation of the device driver at system initialization.

Notification Routine Considerations

- The notification routine (notify_rtn) is called by the monitor dispatcher when a data record has been placed in the device driver's monitor chain buffer. The monitor dispatcher sets ES:SI = address of the device driver's monitor chain buffer and DS = the device driver's DS before calling the notification routine.
- The device driver must process the contents of the monitor chain buffer before returning to the monitor dispatcher. This entry point will be called in the OS/2 mode only.

- When the notify_rtn is called, the first word of the buffer is filled in with the length of the record just sent to the device driver. There is one notification routine call per record.

Purpose

MonWrite passes data records to the monitors for filtering.

Processing

```
LDS  SI,data_record_offset ; Offset of data record in DS
MOV  CX,count              ;Byte count of data record
MOV  AX,monitor_handle     ;handle for chain returned from
                                        ; previous MonitorCreate call
MOV  DH,wait_flag          ;wait/nowait flag
MOV  DL,DevHlp_MonWrite
CALL [Device_Help]
```

Results

```
'C' Clear if OK
    AX = 0
'C' Set if error
    AX = error code
         Invalid monitor handle
         Not enough memory
         Data record too large
```

Remarks

This function may be called at task time or interrupt time. The wait_flag is set to 0 if the MonWrite request occurs at task or user time and the device driver wishes to have the monitor dispatcher perform the synchronization. A value of 1 is specified if no wait is required. A value of 1 must be specified at interrupt time.

The DS register must be set to the device driver data segment.

MonWrite
Give Data to Monitors

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, an overrun has occurred. If it occurs at task (or user) time, the process can block.

A MonFlush in-progress can also cause a not-enough-memory condition.

Each call to MonWrite will send a single complete record. The data sent by this call is considered to be a whole record. A data record must not be larger than the length of the device driver's monitor chain buffer minus two bytes.

The state of the interrupt flag is not preserved across calls to this DevHlp.

Purpose

This function converts a 32-bit address to a GDT selector-offset pair.

Processing

```
MOV  AX,address_high          ;32-bit physical address
MOV  BX,address_low           ;
MOV  CX,length                ;Length of segment
MOV  SI,selector              ;Selector to be setup
MOV  DL,DevHlp_PhysToGDTSelector
CALL [Device_Help]
```

Results

```
'C' Cleared if successful
'C' Set if error
    AX = error code
         Invalid address
         Invalid selector
```

Remarks

PhysToGDTSelector is used to provide addressability through a GDT selector to data. A bimodal device driver supports both real mode and protect mode I/O and must be able to transfer data without being dependent on the current mode of operations. In addition, the interrupt handler of a bimodal device driver must be able to address data buffers regardless of the context of the current process (the current LDT will not necessarily address the data space that contains the data buffer that the interrupt handler needs to access). The GDT selector's addressability will remain valid and the same until another PhysToGDTSelector call is made for the same selector.

The AllocGDTSelector function is used at INIT time to allocate the GDT selectors that the device driver may use with the PhysToGDTSelector. The RealToProt function (see "RealToProt Change Mode from Real to Protect Mode" on page 8-58) is used to change processor mode from real mode to protect mode, and the

PhysToGDTSelector
Map Physical Address to a GDT Selector

ProtToReal function (see "ProtToReal Change Mode from Protect to Real Mode" on page 8-49) is used to return the processor to real mode. The device driver must always return the processor to the same mode it was in when entered if either the ProtToReal or RealToProt functions are used.

PhysToGDTSelector creates selector:offset addressability for a 32-bit physical address. The selector created, however, does not represent a normal memory segment such as those usually managed by OS/2 and is more of a "fabricated segment" for private use by the device driver. Such a segment can not be passed on system calls and may only be used by the device driver to fetch data.

Purpose

In the OS/2 mode PhysToUVirt converts a 32-bit physical address to a valid selector-offset pair addressable out of the current LDT. In the DOS mode, PhysToUVirt converts a 32-bit physical address to a valid segment-offset pair, if the address is below 1 megabyte.

Processing

```
MOV  AX,address_high  ;32-bit physical address
                      ;(or selector, if request type 2)
MOV  BX,address_low   ;
MOV  CX,length        ;Length of area (less than or
                      ; equal to 65535, 0 = 65536)
MOV  DH,request_type  ;Type of Request
                      ; 0 - get virtual address, make
                      ;     segment readable/executable
                      ; 1 - get virtual address, make
                      ;     segment readable/writable
                      ; 2 - free virtual address
                      ;     (OS/2 mode only)
MOV  DL,DevHlp_PhysToUVirt
CALL [Device_Help]
```

Results

```
'C' Set if error
    Invalid address
'C' Cleared if successful
    ES:BX -- segment/selector-offset pair
     (for request types 0 and 1).
```

Remarks

PhysToUVirt will leave its result in ES:BX. PhysToUVirt can also be used in the OS/2 mode to free a selector returned on a prior PhysToUVirt call.

This function is typically used to provide a caller of a device driver with addressability to a fixed memory area, like ROM code and data.

PhysToUVirt
Map Physical To User Virtual Address

The device driver must know the physical address of the memory area to be addressed.

In the OS/2 mode, the offset returned in BX is 0 for request types 0 and 1.

For request type 2, AX contains a selector on entry to PhysToUVirt, and BX and CX are ignored.

PhysToUVirt creates selector:offset LDT addressability for a 32-bit physical address. This function is provided so a device driver can give an application process addressability to a fixed memory area, such as in the BIOS-reserved range from 640Kb to 1Mb. It can also be used to give a client application addressability to a device driver's data segment.

The selector created, however, does not represent a normal memory segment such as those usually managed by OS/2 and is more of a fabricated segment for private use between a device driver and an application. Data within such a segment cannot be passed on system calls and may only be used by the receiving application to fetch data variables.

Map Physical Address to Virtual Address

Purpose

In the OS/2 mode, PhysToVirt converts a 32-bit address to a valid selector-offset pair. In the DOS mode, PhysToVirt converts a 32-bit address to a segment-offset pair.

Processing

```
MOV  AX,address_high          ;32-bit physical address
MOV  BX,address_low           ;
MOV  CX,length                ;Length of segment
MOV  DH,result                ;Leave result
                              ; 0 in DS:SI
                              ; 1 in ES:DI
MOV  DL,DevHlp_PhysToVirt
CALL [Device_Help]
```

Results

```
'C' Cleared if successful
'C' Set if error
    AX = Error code
DH  Set to 0 on input
    DS:SI Valid virtual address
    ES    No mode switch, ES is preserved.
          Mode switch, if ES contains the address of the
           device driver data segment on input, it will
           be converted to a valid virtual address.
           Otherwise, it is set to zero.
DH   Set to 1 on input
     ES:DI  Valid virtual address
     DS   No mode switch, DS is preserved.
          Mode switch, if DS contains the address of the
           device driver data segment on input, it will
           be converted to a valid virtual address.
           Otherwise, it is set to zero.
'Z'  Cleared if no change in addressing mode
'Z'  Set if addressing mode has changed
      previously stored addresses
      must be recalculated.
```

PhysToVirt
Map Physical Address to Virtual Address

Remarks

PhysToVirt will leave its result in either ES:DI or DS:SI, giving the device driver the ability to move strings in either direction. The returned virtual address will not remain valid if the device driver blocks or yields control. The returned virtual address also may be destroyed if the device driver routine which issues the PhysToVirt calls another routine.

On the Personal Computer AT and Personal Computer XT Model 286, if the current mode is the DOS mode and the data lies above 1 Mb, then PhysToVirt will set the target segment register (DS or ES) with a special value and return to the device driver with interrupts disabled. An UnPhysToVirt DevHlp call must be made prior to exiting the device driver in this case.

The device driver must not enable interrupts or change the segment register (ES or DS - whichever contains the returned value) before the device driver has finished accessing the data area. Any change to the contents of the segment register in question will invalidate the mapping. For example, saving and restoring the value in the segment register will cause the register to refer to memory in the first megabyte of system memory. Once the device driver has finished accessing the data area, it must restore the previous interrupt state.

While pointer(s) generated by this routine are in use, the device driver may only call another PhysToVirt request. No other DevHlp routines can be called, because they may not preserve the special DS/ES values created by the PhysToVirt.

The performance characteristics of PhysToVirt are highly variable. In the DOS mode where the entire data area lies below 1 MB or in the OS/2 mode, PhysToVirt is very fast. In the DOS mode where part or all of the data area lies above 1 Mb, PhysToVirt will be very slow.

On the PS/2, if the current mode is the DOS mode and the data lies above 1 Mb, then PhysToVirt will switch into the OS/2 mode and return a selector-offset pair. This mode switch requires the use of the companion function, UnPhysToVirt, to switch back to the DOS mode.

UnPhysToVirt is required to be used any time PhysToVirt is used, with certain qualifications:

1. When use of the converted addresses is ended (no more PhysToVirts), and
2. *Before* the procedure which issued the PhysToVirt returns to its caller.

In addition, multiple PhysToVirt calls may be performed prior to issuing the UnPhysToVirt. Only one call to UnPhysToVirt is needed. When calling PhysToVirt the first time, DS must point to the device driver's device header.

PhysToVirt preserves the registers CS, SS, SP, and DS if called with DH=1, or ES if called with DH=0. The only exception to this case is when a mode switch takes place. If a switch to the OS/2 mode occurs because the specified address lies above 1 Mb and the current mode was the DOS mode, then:

- The segment addresses in CS and SS will be set for the current mode.
- SP will be preserved.
- DS will be preserved unless it is being used for the converted address.
- ES will be set for the current mode only if it contains the data segment value of the device driver and is not being used for the converted address.

If that a previous PhysToVirt had been done (using either DS or ES) with no address mode change flagged, then the PhysToVirt that requires a mode switch to the OS/2 mode causes the previously converted PhysToVirt address to be invalid for the current mode. PhysToVirt must be re-issued to recalculate the address.

When PhysToVirt is being used to recalculate an address after a mode switch is flagged, the second PhysToVirt will not cause a mode switch. The previous address will therefore be valid and preserved (so long as the recalculation uses the opposite segment register as the one that originally caused the mode switch).

PhysToVirt
Map Physical Address to Virtual Address

Note also that in the event of a mode switch, any previously stored address pointers that contain the DS for the device driver's data segment must be stored again by the device driver. The zero flag (ZF) is set if a change in address mode occurred. In this case the device driver must recalculate and store again any buffer addresses that were previously saved.

The pool of temporary selectors used by PhysToVirt in the OS/2 mode is not dynamically extendable. The converted addresses are valid as long as the device driver does not relinquish control (Block, Yield, or RET). An interrupt handler may use converted addresses prior to its EOI, with interrupts enabled. Interrupt handlers should issue an UnPhysToVirt if necessary before making the EOI statement. If an interrupt handler needs to use converted addresses after its EOI, it must protect the converted addresses by running with interrupts disabled.

Note that the task-time strategy routine of a device driver may run enabled on a PS/2.

The segment length parameter must be set to the length of the transfer.

Note: 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 (DS).

For examples on how to use PhysToVirt, see the examples discussed in the section "Using PhysToVirt and UnPhysToVirt" on page 7-68.

Purpose

This DevHlp allows a device driver to change processor mode from protect mode to real mode at interrupt time.

Processing

```
MOV  DL,DevHlp_ProtToReal
CALL [Device_Help]
```

Results

```
'C' Cleared if successful
    Mode has changed to real mode.
'C' Set if error
    Mode has not been changed.
```

Remarks

On entry, DS should be set to the device driver's data segment. On exit, the contents of ES have been changed.

This function is used at interrupt-time to change the processor mode from protect mode to real mode in order to service the device interrupt. The function DevHlp_RealToProt is used to switch the processor mode back from real mode to protect mode (see "RealToProt Change Mode from Real to Protect Mode" on page 8-58).

Once the processing is complete, the device driver must return the processor to the original mode the processor was in when the device driver was entered. For example:

ProtToReal
Change Mode from Protect to Real Mode

```
        .
        .
        .

Device driver entered in Protect Mode.

Call DevHlp_ProtToReal - change to real mode.
        .
        .
        .

Device processing in Real Mode
        .
        .
        .

Call DevHlp_RealToProt - change back to protect mode.

Device driver returns or exits.
```

The following shows an example of how to determine whether the processor must be switched from protect mode to real mode and back by checking the MSW:

```
            smsw   ax          ; get current msw
            push   ax          ; save original msw
            rcr    ax,1        ; pe bit -> cf
            jnc    rm1         ; already real mode?
            DevHlp ProtToReal  ; no, switch to real mode
        rm1: .
            .                  ; process in real mode
            .
            pop    ax          ; get original msw
            rcr    ax,1        ; pe bit -> cf
            jnc    rm2         ; originally in real mode?
            DevHlp RealToProt  ; no, switch back to protect mode
        rm2: .
            .
```

Purpose

PullParticular pulls the specified packet from the selected request packet linked list. If the packet is not found, then an indicator is set on return.

Processing

```
MOV  SI,OFFSET DS:queue        ;Location of queue head (should
                               ; match PushRequest value)
LES  BX,request_packet         ;Pointer to request packet.
MOV  DL,DevHlp_PullParticular
CALL [Device_Help]
```

Results

'C' Cleared if no error.
'C' Set if the specified request is not found.

Remarks

A device driver uses PushReqPacket/PullReqPacket to maintain a work queue for each of its devices. PullParticular is used to remove a specific request packet from the work queue, typically for the case where a process has terminated before finishing its I/O.

PullParticular may also be used to remove request packets that were allocated by an AllocReqPacket from the request packet linked list.

PullReqPacket
Remove Request From List

Purpose

PullReqPacket pulls the next waiting request packet from the selected request packet linked list. If there is no packet in the list, then an indicator is set on return.

Processing

```
MOV  SI,OFFSET DS:queue        ;Location of queue head (should
                               ; match PushReqPacket value)

MOV  DL,DevHlp_PullReqPacket
CALL [Device_Help]
```

Results

```
'C' Set if there is no request.
'C' Cleared if no error.
    ES:BX  Pointer to request packet.
```

Remarks

A device driver uses PushReqPacket/PullReqPacket to maintain a work queue for each of its devices/units. The device driver must provide the storage for the DWORD work queue head, which defines the start of the request packet linked list. The work queue head must be initialized to 0.

PullReqPacket may also be used to remove request packets that were allocated by an AllocReqPacket from the request packet queue.

Purpose

PushReqPacket adds the current device request packet to the linked list of packets to be executed by the device driver.

Processing

```
MOV  SI,OFFSET DS:queue      ;Location of the DWORD queue head
                             ; (which points to the first
                             ; request in the list.)
LES  BX,request_packet       ;Pointer to request
                             ; packet.

MOV  DL,DevHlp_PushReqPacket
CALL [Device_Help]
```

Results

None

Remarks

A device driver uses PushReqPacket/PullReqPacket to maintain a work queue for each of its devices. The device driver must provide the storage for the DWORD work queue head, which defines the start of the request packet linked list. The work queue head must be initialized to 0.

The device driver task-time thread should add all incoming read/write requests to its request list. The driver task-time thread should then determine whether the interrupt-time thread is active, and if not, it should send the request to the device. Because the device may be active at this point, the driver 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 also be used to place request packets that were allocated by an AllocReqPacket in the request packet work queue.

QueueFlush
Clear Character Queue

Purpose

QueueFlush clears the character queue structure that is specified (it empties the buffer).

Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure
                             ; to be flushed.  (The Qsize
                             ; field must be set up.)
MOV  DL,DevHlp_QueueFlush
CALL [Device_Help]
```

Results

None

Remarks

QueueFlush operates on the simple character queue structure initialized by QueueInit.

Purpose

QueueInit initializes the specified character queue structure.

Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure
                             ; to be initialized.  (The
                             ; Qsize field must be set up.)
MOV  DL,DevHlp_QueueInit
CALL [Device_Help]
```

Results

None

Remarks

QueueInit must be called before any other queue manipulation subroutine.

Prior to this call, the device driver must allocate the character queue buffer with the following queue header and initialize the Qsize field.

For example:

```
Qsize     DW    ?    ;size of queue in bytes
Qchrout   DW    ?    ;index of next char out
Qcount    DW    ?    ;count of chars in the queue
Qbase     DB    ?    ;start of queue buffer
```

QueueRead
Read a Character From a Queue

Purpose

QueueRead returns and removes a character from the beginning of the specified character queue structure. If the queue is empty, an indicator is set.

Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure.
MOV  DL,DevHlp_QueueRead
CALL [Device_Help]
```

Results

```
'C' Set if the queue is empty,
'C' Cleared otherwise.
AL  The character read from the queue.
```

Remarks

QueueRead operates on the simple character queue structure initialized by QueueInit.

Purpose

QueueWrite adds a character at the end of the specified character queue structure. If the queue is full, an indicator is set.

Processing

```
MOV  BX,OFFSET DS:queue      ;Points to the queue structure.
MOV  AL,char                 ;Character to insert at the end
                             ; of the queue.
MOV  DL,DevHlp_QueueWrite
CALL [Device_Help]
```

Results

```
'C' Clear if character stored successfully.
'C' Set if queue is full.
```

Remarks

QueueWrite operates on the simple character queue structure initialized by QueueInit.

RealToProt
Change Mode from Real to Protect Mode

Purpose

This DevHlp allows a device driver to change processor mode from real mode to protect mode at interrupt time.

Processing

```
MOV  DL,DevHlp_RealToProt
CALL [Device_Help]
```

Results

```
'C' Cleared if successful
    Mode has changed to protect mode.
'C' Set if error
    Mode has not been changed.
```

Remarks

On entry, DS should be set to the device driver's data segment. On exit, the contents of ES have been changed.

This function is used at interrupt-time to change the processor mode from real mode to protect mode in order to process the device interrupt. The function DevHlp_ProtToReal is used to switch the processor mode back from protect mode to real mode (see "ProtToReal Change Mode from Protect to Real Mode" on page 8-49).

Once the processing is complete, the device driver must return the processor to the original mode that the processor was in when the device driver was entered. For example:

```
        .
        .
        .
Device driver entered in Real Mode.

Call DevHlp_RealToProt - change to protect mode.
        .
        .
        .

Device processing in Protect Mode
        .
        .
        .

Call DevHlp_ProtToReal - change back to real mode.

Device driver returns or exits.
```

The following shows an example of how to determine whether the processor must be switched from real mode to protect mode and back by checking the MSW:

```
     smsw   ax          ; get current msw
     push   ax          ; save original msw
     rcr    ax,1        ; pe bit -> cf
     jc     pm1         ; already protect mode?
     DevHlp RealToProt  ; no, switch to protect mode
pm1: .
     .                  ; process in protect mode
     .
     pop    ax          ; get original msw
     rcr    ax,1        ; pe bit -> cf
     jc     pm2         ; originally in protect mode?
     DevHlp ProtToReal  ; no, switch back to real mode
pm2: .
     .
```

Register
Add Monitor

Purpose

Register adds a monitor to the chain of monitors for a class of device.

Processing

```
LES  SI,input_buffer          ;Address of input buffer
MOV  DI,output_buffer_offset ;Offset of output buffer
MOV  CX,monitor_PID           ;Process ID of monitor task
MOV  AX,monitor_handle        ;handle for chain returned from
                                        ; previous MonitorCreate cal
MOV  DH,placement_flag        ;High or Low place in chain
MOV  DL,DevHlp_Register
CALL [Device_Help]
```

Results

```
'C' Clear no error
'C' Set if error
    AX = error code
         Invalid monitor handle
         Not enough memory
```

Remarks

This function may only be called at task time in the OS/2 mode.

A monitor chain must have previously been created with MonitorCreate.

A single process may register more than one monitor (with "different" input and output buffers) with the same monitor chain.

The first word of each of the input and output buffers must contain the length in bytes (length word inclusive) of the buffers. The length of the monitor's input and output buffers must be greater than the length of the device driver's buffer plus 20 bytes.

The input buffer, output buffer offset, and placement flag are supplied to the device driver by the monitor task (which is asking to be registered).

The device driver must identify the monitor chain with the monitor handle returned from a previous MonitorCreate call. The device driver can determine the PID of the requesting monitor task from the Local InfoSeg. Refer to "GetDOSVar Get Address of Important DOS Variables" on page 8-29.

ResetTimer
Reset Timer Handler

Purpose

ResetTimer removes a timer handler for the device driver.

Processing

```
MOV  AX,Offset cs:timer_handler        ;offset to timer handler
MOV  DL,DevHlp_ResetTimer
CALL [Device_Help]
```

Results

```
'C' Cleared if no error
'C' Set if error
    Invalid handler
```

Remarks

This function removes a timer handler from the list of timer handlers. Timer handlers are analogous to the user timer interrupt (Int 1CH). Refer to "TickCount Modify timer" on page 8-84 for a discussion on timer handlers.

DS should be set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it should restore DS to the original value.

Purpose

ROMCritSection flags a critical section of execution in a DOS mode software interrupt handler to prevent the DOS mode from being suspended in the background.

Processing

```
MOV  AL,enter_or_exit          ;Critical Section flag
                               ; = 0  exit
                               ; < > 0  enter
MOV  DL,DevHlp_ROMCritSection
CALL [Device_Help]
```

Results

None.

Remarks

This service is called by a device driver's DOS mode software interrupt handler.

Sections of ROM BIOS code must be protected from preemption. Preemption occurs when a user switches away from the DOS mode. This causes the DOS mode to be suspended in background. However, some I/O processing 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 OS/2 device driver to intercept the appropriate ROM BIOS interrupt and issue the DevHlp function call, ROMCritSection, to protect the ROM BIOS critical section of execution.

ROMCritSection
Flag Critical Section of Execution

Note: When the OS/2 device driver issues ROMCritSection to "enter" a ROM BIOS critical section, the user will not be able to switch away from the screen of the DOS mode to another screen. This has potential problems for the user. For example, if some DOS 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 the user will be unable to switch tasks. The worst case is that the terminate-and-stay-resident application is interactive, never allowing the OS/2 device driver to issue the "exit" from critical section and never allowing the user to switch away from the screen of the DOS mode until the user terminates the application.

Purpose

This is the companion routine to Block. When Run is called, it awakens *all* threads that were blocked for this particular event identifier.

Processing

```
MOV  BX,event_id_low          ;Low word of event identifier.
MOV  AX,event_id_high         ;High word of event identifier.
MOV  DL,DevHlp_Run
CALL [Device_Help]
```

Results

```
AX = count of those woken.
'Z'  set according to AX.
```

Remarks

Run returns immediately to its caller; the awakened threads will be run at the next available opportunity. Run is often called at interrupt time.

Refer to "Block This Thread From Running" on page 8-19 for a detailed discussion.

SchedClockAddr
Get system clock routine

Purpose

This service is provided to the clock device driver to allow it to obtain a pointer to the address of the system's clock tick handler, SchedClock. SchedClock must be called on each occurrence of a periodic clock tick.

Processing

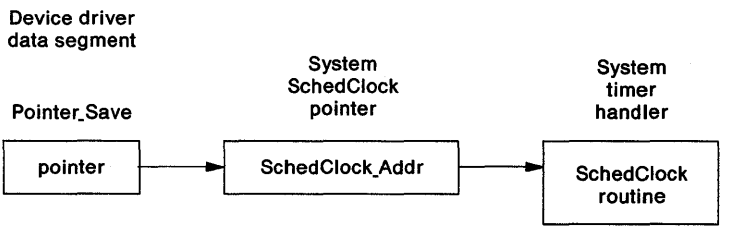
```
MOV  AX,Pointer_Save              ;Offset in DS to a DWORD
                                  ; where the pointer will
                                  ; be returned
MOV  DL,DevHlp_SchedClockAddr
CALL [Device_Help]
```

Results

```
Pointer_Save will contain the address of
the system tick handler.
```

Remarks

The clock device driver calls this DevHlp service during the clock device driver's initialization. For input to this DevHlp, the clock device driver must ensure that its DS points to the device driver's data segment and supply the offset to a DWORD area. The DevHlp service will then fill in the device driver's save area with a bimodal pointer to a DWORD in system memory. The DWORD in system memory contains the pointer to the SchedClock routine. This is illustrated in the following diagram:



The pointer that is returned to the device driver in Pointer_Save is a bimodal pointer which is valid in either the DOS mode or the OS/2 mode. The device driver does not need to perform any conversions on this pointer for DOS mode or OS/2 mode operations. The device driver can then use the pointer it has in Pointer_Save to call SchedClock. SchedClock is called by the clock device driver with a CALL FAR INDIRECT, using the pointer to the area which contains the actual address of the SchedClock routine.

SchedClock must be called at interrupt time for each periodic clock tick to indicate the passage of system time. The "tick" is then dispersed to the appropriate components of the system. A definition of the interface to SchedClock follows.

```
MOV  AL,millisecs              ;Milliseconds since last call
MOV  DH,EOIflag                ;Indicator of EOI
                               ; = 0 prior to EOI
                               ; = 1 after EOI
CALL [pointer]                 ;Pointer to the area which
                               ; contains the actual address
                               ; of SchedClock
```

The clock device driver's interrupt handler must run with interrupts enabled as the convention, 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 SchedClock. 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. The DevHlp SchedClock is also reentrant.

The device driver must not get the actual address of the SchedClock routine but instead use the pointer returned by the DevHlp call.

SemClear
Release a Semaphore

Purpose

This function releases a semaphore and restarts any blocked threads waiting on the semaphore.

Processing

```
MOV  BX,sem_handle_low     ;Semaphore handle
MOV  AX,sem_handle_high    ;
MOV  DL,DevHlp_SemClear
CALL [Device_Help]
```

Results

```
'C' Cleared if no error
'C' Set if error
    AX = error code
         error_invalid _handle
         error_excl_sem_already_owned
         error_invalid_at_interrupt_time
         error_protection_violation
```

Remarks

A device driver may clear either a RAM semaphore or a system semaphore. The device driver may obtain (own) a semaphore through SemRequest.

The semaphore handle for a RAM semaphore is the virtual address of the doubleword of storage allocated for the semaphore. Virtual address is used as a generic term for addresses: segment:offset for the DOS mode; selector:offset for the OS/2 mode. To access a RAM semaphore at interrupt time, the device driver must locate the semaphore in the device driver's data segment (DS).

For a system semaphore, the handle must be passed to the device driver by the caller by way of a generic IOCtl. The device driver must convert the caller's handle to a system handle with SemHandle.

SemClear
Release a Semaphore

A RAM semaphore can be cleared at interrupt time only if it is in storage that is directly addressable by the device driver, that is, in the device driver's data segment.

SemHandle
Obtain a Semaphore Handle

Purpose

This function provides a semaphore handle to the device driver.

Processing

```
MOV  BX,sem_key_low     ;Semaphore identifier
MOV  AX,sem_key_high    ;
MOV  DH,usage_flag      ;Indicates if in use
                        ; = 0 not-in-use
                        ; = 1 in-use
MOV  DL,DevHlp_SemHandle
CALL [Device_Help]
```

Results

```
'C' Cleared if no error
    AX:BX set to the system handle.
'C' Set if error
    AX contains error code.
    Invalid handle for the semaphore if DH = 1.
```

Remarks

This function is used to convert 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. This allows the System Semaphore to be referenced at interrupt time by the device driver. This "key" is also used when the device driver is finished with the system semaphore. The device driver must call SemHandle with the usage_flag indicating that the device driver is finished with the system semaphore.

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 DOS mode and OS/2 mode address forms (segment:offset, selector:offset). SemHandle may be used for RAM semaphores. Because RAM semaphores have no system handles, SemHandle will simply return the RAM semaphore "key" back to the caller.

A device driver can determine that a semaphore is a RAM semaphore if the key remains unchanged upon return from the SemHandle function. If the key returned from SemHandle is different than the one passed to the function, then the device driver can determine that it is a handle for a System semaphore.

If carry is returned from this function, the device driver should issue the DevHlp VerifyAccess request with TypeOfAccess of Read/Write indicated before assuming this is a RAM semaphore. If a Ram semaphore is to be used, it must be accessed only at task time unless it is in locked storage.

It is necessary to call SemHandle at task time to indicate that a System Semaphore is IN-USE because:

1. 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 (task time or interrupt time).
2. 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 used by the original semaphore could be reassigned. A device driver which tried to manipulate the original process's semaphore would inadvertently manipulate the new process's semaphore. Therefore, the SemHandle IN-USE indicator increases a counter so that, although the calling thread may still delete its task-specific refer-

ence to the semaphore, the semaphore remains in the system structures.

A device driver must subsequently call SemHandle with NOT-IN-USE when the semaphore use is done so that the system semaphore structure can be freed. There must be a call to indicate NOT-IN-USE to match every call to indicate IN-USE (one-to-one relationship).

The state of the interrupt flag is not preserved across calls to this DevHlp.

Purpose

This function claims a semaphore. If the semaphore is already owned, the thread in the device driver is blocked until the semaphore is released or until a timeout occurs.

Processing

```
MOV  BX,sem_handle_low    ;Semaphore handle
MOV  AX,sem_handle_high   ;
MOV  CX,sem_timeout_low   ;Timeout value
MOV  DI,sem_timeout_high  ;in milliseconds
                          ; = -1 wait forever
                          ; =  0 no wait if sem owned
                          ; >  0 timeout
MOV  DL,DevHlp_SemRequest
CALL [Device_Help]
```

Results

```
'C' Cleared if no error.
'C' Set if error.
    AX = error code.
         error_sem_timeout
         error_sem_owner_died
         error_invalid_handle
         error_too_many_sem_requests
         error_interrupted
         error_protection_violation
```

Remarks

SemRequest checks the state of the semaphore. If it is unowned, SemRequest marks it "owned" and returns immediately to the caller. If the semaphore is owned, SemRequest will optionally block the device driver thread until the semaphore is unowned, then try again. The timeout parameter is used to place an upper limit on the amount of time to block before returning to the requesting device driver thread.

SemRequest
Claim a Semaphore

SemClear is used at either task time or interrupt time to release the semaphore.

The semaphore handle for a RAM semaphore is the virtual address of the doubleword of storage allocated for the semaphore. Virtual address is used as a generic term for addresses: segment:offset for the DOS mode; selector:offset for the OS/2 mode. To access a RAM semaphore at interrupt time, the device driver must locate the semaphore in the device driver's data segment (DS).

For a system semaphore, the handle must be passed to the device driver by the caller through a generic IOCtl. The device driver must convert the caller's handle to a system handle with SemHandle.

Note that this service is valid in user mode only for RAM semaphores. System semaphores are not available in user mode by device drivers.

The state of the interrupt flag is not preserved across calls to this DevHlp.

Purpose

This routine is called by a device driver to indicate the occurrence of an event.

Processing

```
MOV  AH,event                ;Event being signalled
MOV  BX,argument             ;Parameter for the event
                             ; being signalled
MOV  DL,DevHlp_SendEvent
CALL [Device_Help]
```

Results

```
'C' Cleared if no error.
'C' Set if error sending signal.
```

Remarks

The events are defined as:

- Event = 0 (Reserved)
- Ctrl + Break
  - event = 1
  - argument = 0 (Reserved)
- Ctrl + C
  - event = 2
  - argument = 0 (Reserved)
- Ctrl + NumLock
  - event = 3
  - argument = Foreground session number

SendEvent
Indicate an Event

- Ctrl + PrtSc
  - event = 4
  - argument = 0 (Reserved)
- Shift + PrtSc
  - event = 5
  - argument = 0 (Reserved)
- Session Manager Hot Key from the Keyboard
  - event = 6
  - argument = Hot Key ID

  The keyboard device driver uses the Hot Key ID which was set by way of keyboard IOCtl 56H (SET SESSION MANAGER HOT KEY).

Purpose

This service is used to set a hardware interrupt vector to the device driver interrupt handler.

Processing

```
MOV  AX,Offset CS:handler ;Interrupt handler offset
MOV  BX,IRQnum            ;Interrupt level number (0 - FH)
MOV DH,Shared_Int         ;Interrupt sharing (=0 not shared, =1 shared)
MOV  DL,DevHlp_SetIRQ
CALL [Device_Help]
```

Results

```
'C' Clear if no error
'C' Set if error
       IRQ is not available.
```

Remarks

The attempt to register an interrupt handler for an IRQ to be Shared will fail if the IRQ is already owned by another device driver as Not Shared, owned by a DOS mode interrupt handler, or is the IRQ used to cascade the slave 8259 interrupt controller (IRQ 2).

The attempt to register an interrupt handler for an IRQ to be Not Shared will fail if the the IRQ is already owned by another device driver as Shared or Not Shared, owned by a DOS mode interrupt handler, or is the IRQ used to cascade the slave 8259 interrupt controller.

DS should be set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it should restore DS to its original value.

The IRQnum value is range checked and 'C' is set if not from 0 to 0FH.

SetROMVector
Set DOS Mode Software Interrupt Vector

Purpose

This service is used to replace a DOS mode software interrupt handler with a handler from the device driver. This service returns a DOS mode pointer to the previous software interrupt handler for chaining.

Processing

```
MOV  AX,OFFSET CS:handler    ;Interrupt handler offset
MOV  BX,intnum               ;Interrupt number
MOV  SI,OFFSET CS:saveDS     ;offset in CS of WORD
                             ; to save DOS mode DS
MOV  DL,DevHlp_SetROMVector
CALL [Device_Help]
```

Results

```
'C' Clear if no error
    AX:DX = DOS mode pointer to previous header.
    CS:[SI] = DOS mode DS of device driver.
'C' Set if error
    Invalid interrupt number.
```

Remarks

The device driver can register a software interrupt handler for a DOS mode software interrupt. This is typically done to intercept a ROM BIOS software interrupt, which allows the device driver to perform any processing needed to coordinate device I/O between the driver and ROM BIOS.

The device driver may not register a software interrupt handler for any of the interrupt vectors reserved for hardware interrupts. The reserved interrupt numbers are:

08 - 0FH
50 - 57H
70 - 77H

Set DOS Mode Software Interrupt Vector

The device driver's software interrupt handler for the DOS mode software interrupt receives control directly when the interrupt occurs. Consequently, the DS register is not set up for the handler on entry. The handler must set the DS register with the value that SetROMVector had previously saved in the CS:[SI] location.

When calling SetROMVector, the device driver must ensure that DS is set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it must restore DS to the original value.

SetTimer
Set Timer Handler

Purpose

SetTimer adds a timer handler to the list of timer handlers to be called on a timer tick.

Processing

```
MOV  AX,OFFSET CS:timer_handler        ;Offset of timer handler.
MOV  DL,DevHlp_SetTimer
CALL [Device_Help]
```

Results

```
'C' Cleared if no error.
'C' Set if error.
    Timer handler disallowed (maximum number of
     handlers reached or timer handler already set).
```

Remarks

The DevHlp SetTimer is a subset of the DevHlp TickCount.

This function allows a device driver to add a timer handler to a list of timer handlers called on every timer tick. A device driver may use a timer handler to drive a non-interrupt device instead of using time-outs with the Block and Run services. Block and Run are costly on a character-by-character basis; the cost is one or more task switches per character I/O. Timer handlers are required to save and restore registers.

A maximum of 32 different timer handlers are available in the system.

While a timer handler is in the format of a FAR CALL/RETURN routine (when it is finished processing, it performs a return), it operates in Interrupt State. The timer handler is analogous to the user timer (Int 1CH) handler. Care should be taken not to remain in the handler very long.

DS should be set to the device driver's data segment. If the device driver had done a PhysToVirt referencing the DS register, it should restore DS to the original value.

Timer handlers are responsible for saving and restoring registers on entry and exit.

SortReqPacket
Insert Request In Sorted Order To List

Purpose

This routine is used by block (disk) device drivers to add a new request to their work queue. This routine inserts the request packet in the linked list of request packets in the order of starting sector number.

Processing

```
MOV  SI,OFFSET DS:queue      ;Location to DWORD queue
                             ; head (which points to
                             ; first request).  It
                             ; should be initialized
                             ; to 0.
LES  BX,request_packet       ;Pointer to request
                             ; packet.
MOV  DL,DevHlp_SortReqPacket
CALL [Device_Help]
```

Results

None

Remarks

The sorting by sector number is aimed at reducing the length and number of head seeks. This is a simple algorithm and does not account for multiple heads on the media or for target drive in the request packet. SortReqPacket inserts the current request packet into the specified linked list of packets, sorted by starting sector number.

SortReqPacket may be used to place request packets that were allocated by an AllocReqPacket in the request packet queue.

Purpose

This function is similar to the Yield function, except that the CPU may only be yielded to a time-critical thread, if one is available.

Processing

```
MOV  DL,DevHlp_TCYield
CALL [Device_Help]
```

Results

None

Remarks

It is not necessary for the device driver to do both a Yield and a TCYield; the TCYield function is a subset of the Yield function.

The one part of the kernel that can take a lot of CPU time is in device drivers, particularly those that perform program I/O on long strings of data or that poll a device. These device drivers periodically should check the TCYield Flag and call the TCYield function to yield the CPU to a time-critical thread.

The location of the TCYield Flag is obtained from the GetDOSVar call.

For performance reasons, the device driver should check the TCYield Flag once every 3 milliseconds. If the flag is set, then the device driver should call TCYield.

Because the device driver may relinquish control of the CPU, the device driver should not assume that the state of the interrupt flag will be preserved across a call to TCYield.

TickCount
Modify timer

Purpose

TickCount will register a new timer handler or modify a previously registered timer handler to be called on every N timer ticks instead of every timer tick.

Processing

```
MOV AX,OFFSET CS:timer_handler  ;offset to timer handler
MOV BX,count                    ;N tick counts (0-FFFF)
                                ; 0 means FFFFH+1 ticks

MOV  DL,DevHlp_TickCount
CALL [Device_Help]
```

Results

```
'C' Cleared if no error
'C' Set if error
    Timer handler cannot be modified or set.
```

Remarks

A device driver may use a timer handler to drive a non-interrupt device instead of using timeouts with the Block and Run services. Block and Run are costly on a character-by-character basis; the cost is one or more task switches per character I/O. Timer handlers are required to save and restore registers.

While a timer handler is in the format of a FAR CALL/RET routine, it operates at interrupt time. The timer handler is analogous to the user timer (INT 1CH) handler. Care must be taken not to remain in the handler very long.

For a new timer handler, TickCount will register the timer handler to be called on every N 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 will allow device drivers to support the timeout function without needing to count ticks.

At task time, this DevHlp may be used to modify a timer handler registered through SetTimer or may be used to register a new timer handler that is initially invoked every N ticks.

In user mode (task time), this DevHlp may be used only to modify a timer handler *already* registered.

In interrupt mode (interrupt time), this DevHlp may be used only to modify a timer handler *already* registered. This allows an interrupt handler to reset the timing condition at interrupt time.

Note that SetTimer sets a default of N ticks to 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.

DS should be set to the device driver's data segment. If the device driver did a PhysToVirt referencing the DS register, it should restore DS to the original value.

Timer handlers are responsible for saving and restoring registers on entry and exit.

A maximum of 32 different timer handlers are available in the system.

Unlock Memory Segment

Purpose

This service unlocks a locked memory segment.

Processing

```
MOV  BX,lock_handle_low       ;Handle for segment
MOV  AX,lock_handle_high      ; returned by Lock
MOV  DL,DevHlp_Unlock
CALL [Device_Help]
```

Results

```
'C' Cleared if no error.
'C' Set if error.
    Invalid handle or cannot unlock memory.
```

Remarks

UnLock cannot be used on memory allocated by AllocPhys.

Purpose

UnPhysToVirt is *required* to mark completion of address conversion from PhysToVirts.

Processing

```
MOV  DL,DevHlp_UnPhysToVirt
CALL [Device_Help]
```

Results

```
'Z' Cleared if no address mode change.
'Z' Set if address mode change.
```

Remarks

This function forms part of the structure of mode-dependent addressing on behalf of a device driver, relieving it of the need to understand the CPU mode and the subsequent effects on accessing memory.

On the PS/2, if the converted address caused a switch to the OS/2 mode in PhysToVirt, UnPhysToVirt will switch back to the DOS mode.

UnPhysToVirt must be called by the same procedure that issued the PhysToVirt when use of converted addresses is completed and *before* the procedure returns to its caller. The procedure that called PhysToVirt may call other procedures before calling UnPhysToVirt. Multiple PhysToVirt calls may be issued prior to issuing the UnPhysToVirt. Only one call to UnPhysToVirt is needed.

The 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.

UnPhysToVirt
Mark Completion of Virtual Address Use

UnPhysToVirt, if switched to the DOS mode, will reset the registers CS and SS to the DOS mode.

SP will be preserved.

DS will be reset to the device driver's data segment.

Note that the addresses that caused the switch into the OS/2 mode cannot be preserved or converted for the DOS mode.

The ES register will not be preserved. The ES register is also set to the device driver data segment.

Purpose

This routine removes the current hardware interrupt handler.

Processing

```
MOV  BX,IRQnum                ;IRQ Interrupt number
                              ; ( 0 - F )
MOV  DL,DevHlp_UnSetIRQ
CALL [Device_Help]
```

Results

```
'C' Set if caller is not the owner of
     or one of the owners of the IRQ.
```

Remarks

DS must point to the device driver's data segment on entry.

VerifyAccess
Verify Access to Memory

Purpose

This routine verifies that the user process has the correct access rights for the memory that it passed to the device driver.

Processing

```
MOV  AX,Segment_@            ;Selector or segment
MOV  CX,MemLength            ;Length of memory area
                             ; in bytes (0 means 64 KB)
MOV  DI,Mem_Offset           ;Offset to memory area
MOV  DH,TypeOfAccess         ;Verify access for
                             ; = 0  read access
                             ; = 1  read/write access
MOV  DL,DevHlp_VerifyAccess
CALL [Device_Help]
```

Results

```
'C' Clear if no error
    Access verified.
'C' Set if error
    Access attempt failed.
```

Remarks

A device driver can receive addresses to memory as part of a generic IOCtl request from a process. Because the operating system cannot verify addresses imbedded in the IOCtl command, the device driver must request verification in order to prevent itself from accidentally erasing memory on behalf of a user process. If the verification test fails, then VerifyAccess will terminate the process.

Note that verification may only take place in the OS/2 mode. If VerifyAccess is called in the DOS mode, it will return stating that the memory is accessible.

Once the process has been verified as having the needed access to a specific address location, the device driver doesn't need to 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, then 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 with the VerifyAccess DevHlp 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 valid for the duration of the lock.

VirtToPhys
Map Virtual Address to Physical Address

Purpose

When in the OS/2 mode, it converts a selector-offset pair to a 32-bit physical address. When in the DOS mode, VirtToPhys converts a segment-offset pair to a 32-bit physical address.

Processing

```
LDS  SI,address              ;Virtual address:
                             ; segment:offset or selector:offset
MOV  DL,DevHlp_VirtToPhys
CALL [Device_Help]
```

Results

```
'C' Cleared to indicate no error.
    AX:BX  Physical address: 32-bit number.
```

Remarks

The virtual address should be locked by way of the DevHlp service Lock prior to invoking this function, if the segment is not known to be locked already.

This function is typically used to convert a virtual address supplied by a process by way of a generic IOCtl, in order that the memory may be accessed at interrupt time.

Yield
Relinquish the CPU

Purpose

This routine yields the CPU to a scheduled thread of equal or higher priority.

Processing

```
MOV  DL,DevHlp_Yield
CALL [Device_Help]
```

Results

None

Remarks

OS/2 is designed so that the CPU is never preemptively scheduled while in kernel mode. In general, the kernel either performs its job and exits quickly, or it blocks waiting for (usually) I/O or (occasionally) a resource. It is not necessary for the device driver to do both a Yield and a TCYield; the Yield function is a superset of the TCYield function.

The one part of the kernel that can take a lot of CPU time is in device drivers, particularly those that perform program I/O on long strings of data or poll the device. These drivers should periodically check the Yield Flag 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 Yield Flag is obtained from the GetDOSVar call. Refer to "GetDOSVar Get Address of Important DOS Variables" on page 8-29.

Yield
Relinquish the CPU

For performance reasons, the device driver should check the Yield Flag once every 3 milliseconds. If the flag is set, then the device driver should call Yield.

Because the device driver may relinquish control of the CPU to another thread, the device driver should not assume that the state of the interrupt flag will be preserved across a call to Yield.

Chapter 9. Device Drivers

The device drivers provided with OS/2 service requests in both the DOS mode and the OS/2 mode. Where appropriate, OS/2 device drivers provide a queued request interface rather than the serial request design of DOS device drivers. OS/2 device drivers support multitasking.

This chapter references the IOCtl interface. The IOCtl interface allows an application or subsystem to send device-specific control commands to the device driver. The IOCtl function uses the DosDevIOCtl function request for OS/2 applications and the INT 21H IOCtl request for DOS applications. See *Technical Reference, Vol. 2* for a detailed description of the DosDevIOCtl function request and the IOCtl functions.

ASYNC (RS232-C) Communications Device Driver

The Asynchronous Communications (ASYNC) device driver enables application programs or system programs (for example, Print Spooler) to utilize the RS232-C interface of the system in the OS/2 mode. The device driver will allow an application program in the OS/2 mode to support full duplex communications while the device driver:

- Services the RS232-C port in an interrupt-driven manner.
- Provides transmit and receive queues.
- Provides different automatic control modes of the modem control signals.
- Provides logical data stream flow control (XON/XOFF) for transmit and receive.

The ASYNC device driver can be installed optionally by the user via a DEVICE= command in CONFIG.SYS. This device driver uses up low memory, making that memory unavailable to DOS mode programs; therefore, this device driver should not be installed unless it is required.

The user will normally want to use the ASYNC device driver either in conjunction with the Print Spooler (if a serial printer is to be used), or with an application program that utilizes the RS232 enabling capabilities of the ASYNC device driver coupled with a serial device attached to the system.

Hardware Support

Supported hardware for the RS232-C ASYNC communications device driver includes:

- The IBM Personal Computer AT and IBM Personal Computer XT Model 286. The IBM Personal Computer AT Serial/Parallel Adapter card (#0215, #3395, and #3400) is the supported adapter.
- The PS/2 family computers (Models 50, 60 and 80).

Personal Computer AT Adapter Support

On the Personal Computer AT, the device driver supports a maximum of two ASYNC ports, each on separate interrupt levels. The device driver will recognize ASYNC ports with the following base I/O addresses:

- 3F8H (must generate a level 4 interrupt)
- 2F8H (must generate a level 3 interrupt).

No other base I/O addresses will be recognized or supported, and no other interrupt level combinations are supported.

The ASYNC device driver for the Personal Computer AT interfaces directly to the hardware.

PS/2 Adapter Support

The ASYNC device driver for the PS/2 uses the Advanced BIOS interface for the ASYNC device. It does not interface directly to the hardware and does not support any device which requires direct hardware access.

The device driver supports a maximum of three ASYNC ports on a maximum of two different interrupt levels. The interrupt levels must have Advanced BIOS support, with one unit per Logical ID (LID) for the ASYNC device ID.

The only ASYNC devices supported on PS/2 are COM1, COM2, and COM3. These devices correspond to the first three LIDs in the Advanced BIOS common data area that have the architected ASYNC device ID. These devices also correspond to the first three ASYNC addresses in the ASYNC 40: area.

If a device has capabilities other than ASYNC which cannot be utilized independently (for example, as in the Advanced BIOS separate LID architecture, and others) of the ASYNC capabilities, and if Advanced BIOS assigns that device the ASYNC device ID, then that device can only be used for ASYNC in that power-on session.

If the device is not assigned the ASYNC device ID, it is not supported by this device driver.

If the device is assigned the ASYNC device ID and it has additional capabilities (for example, a built in modem) beyond supporting the RS232-C port, the device driver will not recognize those additional capabilities (and potential limitations). Also, the device driver will not inform any application program of those additional capabilities (or limitations). And the device driver will not limit the control of the RS232-C interface or the device to only those modes which are acceptable to the extended hardware capabilities of that RS232-C port.

If an ASYNC device is not supported by OS/2 but is recognized by Advanced BIOS as an ASYNC device ID, the device driver may recognize and try to use that unsupported device if it is COM1, COM2, or COM3.

Attachment Support

The ASYNC device driver does not provide any support for devices attached to the RS232-C port. The device driver provides enabling support for the RS232-C interface itself. Application programs, subsystems, and systems programs provide the support needed to use devices attached to the RS232-C port.

The ability to support a device can be determined by understanding the level of RS232-C interface enabling support the device driver provides, along with the characteristics of the attachment hardware in question and the required functions to be supported.

The ASYNC device driver provides a mechanism where one or more additional device drivers can be installed to support specific COM ports. This feature may be required for the following reasons:

- To allow an application program to support a special device not adequately supportable with this ASYNC device driver.
- To allow additional COM ports (besides COM1-3 on PS/2) to be supported.
- To enhance the level of device driver function for a given COM port. (This may be required for certain subsystem support.)

RS232-C Interface

The ASYNC interface consists of separate read and transmit lines.

There are two separate modem control signals whose output values can be controlled by the device driver:

- Data Terminal Ready (DTR)
- Request To Send (RTS).

There are four separate modem control signals whose input values are available to the device driver:

- Data Set Ready (DSR)
- Clear To Send (CTS)
- Data Carrier Detect (DCD) also known as Receive Line Signal Detect (RLSD)
- Ring Indicator (RI).

The receive and transmit data lines have the following hardware characteristics:

- Logical 1 - Marking - More negative than -3 Volts. This state could mean no data.
- Logical 0 - Spacing - More positive than +3 Volts. This state could mean break condition.

The modem control signal lines have the following hardware characteristics:

- Function ON when more positive than +3 Volts.
- Function OFF when more negative than -3 Volts.

RS232-C Enabling Characteristics

The device driver supports the ASYNC interface in an interrupt-driven manner. This allows the multitasking capabilities of OS/2 to be supported while ASYNC data reception and transmission is taking place.

With the current ASYNC hardware, when data is given to the transmit hardware, data will be transmitted at the physical RS232-C interface. When data is given to the transmit hardware, it has not yet been physically transmitted (at the RS232 interface). The data is considered completely transmitted by the transmit hardware at the physical RS232 interface when the transmit shift register of the UART is empty. The IOCtl Return Transmit Data Status (Category 1 Function 65H) can be used to determine this information.

The device driver transmit queue is a memory buffer between the OS/2 system and the transmit hardware. It is considered to be owned by the device driver because the device driver controls the data movement into and out of the transmit queue. Algorithms for this data movement may change between releases of the device driver. Changes in the ASYNC hardware may cause changes in the data movement algorithms and/or external interfaces.

The device driver receive queue is a memory buffer between the OS/2 system and the receive hardware. It is considered to be owned by the device driver because the device driver controls the data movement into and out of the receive queue. Algorithms for this data movement also can change between releases of the device driver. Changes in the ASYNC hardware can cause changes in the data movement algorithms and/or external interfaces.

Data that applications send (made available by Write requests) get placed in the device driver transmit queue. When an interrupt occurs to tell the device driver that the hardware is ready for more data, the device driver will give the transmit hardware more data from the transmit queue.

When an interrupt occurs to tell the device driver that the hardware has received data, that data is placed in the device driver receive queue. When the device driver gets a read request (Read request packet) from the application, the device driver fills the read request from the receive queue.

The size of the receive and transmit queues are available from the following IOCtls:

- Return number of chars in receive queue (Category 1 Function 68H).
- Return number of chars in transmit queue (Category 1 Function 69H).

The device driver services each communications port independently. Requests issued to a given port have no effect on any other communications ports that the device driver may be servicing.

The device driver processes Read and Write request packets independently for a given port. An application can be written to support simultaneous reception and transmission of data. In addition, the device driver can process an IOCtl request simultaneously with outstanding Read and Write requests.

The device driver does not schedule the processing of IOCtl requests. The device driver processes the IOCtl request when received, regardless of what else it is doing. This may cause unexpected results if, for instance, the baud rate is modified while data reception or transmission is taking place.

The application should issue only one IOCtl request at a time. If the application issues another IOCtl request before the first IOCtl request completes, the results are UNDEFINED.

The device driver will queue multiple Read and Write request packets independently. The device driver always will begin processing the Read request packets in the order that they are received. It will always begin processing the Write request packets in the order that they are received.

Note: The operating system does not guarantee that file system requests will be delivered to a device driver in the order in which they are issued by an application. This means that a request by one

thread can get blocked in the operating system, thus allowing a subsequent request by a different thread for the same function (for example, DosWrite) to pass through and arrive ahead of the first thread at the device driver. This is true for synchronous operations performed by multiple threads, or asynchronous operations performed by the same thread.

Because of thread priority considerations and the system dynamics, the order observed by the application of completing requests of the same type may not be in the order that they were received by the device driver. The device driver will always keep the data in the same order in which the Read and Write request packets (of the same type) were received in. There is no ordering or timing between different types of request packets.

A First Level Open is described in the section on "States of the ASYNC Device Driver" on page 9-11. A First Level Open occurs when the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close). A CLOSE request packet causing the device driver to process the next OPEN request packet as a First Level Open, is called a last level close. Because the requests that an application issues sometimes get out of order before they reach the device driver, an application cannot consider a CLOSE a last level CLOSE until the CLOSE completes. If the application issues an OPEN request to the COM port before a previously issued CLOSE request is completed, then the results are UNDEFINED.

The Flush request may be completed before all the appropriate request packets (that have been queued by the device driver) have been flushed. The appropriate request packets will eventually be flushed (and return to the caller) based on their priority and the system dynamics. Once the Flush request has been processed, the appropriate request packets will not cause data to be transmitted (or received data to be moved) incorrectly.

The device driver supports different time-out processing characteristics and time-out settings for the Read and Write requests.

The device driver removes from an application how to know exactly when a given character is being transmitted or received (at the hardware interface). Therefore, an application cannot expect to provide real-time flow control of data (in the middle of data transmission or

reception) based on logical characters (XON/XOFF) or based on the state of the modem control signals by:

- Manually controlling or monitoring those modem control signals
- Manually monitoring the queue status
- Manually monitoring data moving across the link.

Therefore, the device driver provides automatic modes of operation that are controllable via IOCtls, to allow OS/2 mode applications automatically to control the data flow through the RS232-C port.

Output Modem Control Signals

Besides allowing the application to control directly RTS and DTR, the device driver has different automatic control modes to control the value of the output modem control signals. They are:

- Open and Close processing of DTR and RTS
- Disable/Enable DTR & RTS
- RTS toggling on transmit
- Input handshaking using DTR & RTS.

These different control modes are described in the section on "States of the ASYNC Device Driver" on page 9-11 and in the IOCtls description.

Note: The level of support provided by this device driver requires that DTR and RTS are turned on at least once, even if this puts the device driver in a mode where they will never be turned on again.

Input Modem Control Signals

Besides allowing the application to read directly the current state of DSR, CTS, DCD, and RI, the device driver has different automatic modes that cause it to respond to the value that some input modem control signals may have. They are:

- Output handshaking using CTS, DSR, DCD
- Input sensitivity using DSR.

These different control modes are described in the section on "States of the ASYNC Device Driver" on page 9-11 and in the IOCtls description.

Additional information on the state of the input modem control signals is available by using the IOCtl Return COM Event Information (Category 1 Function 72H).

Logical Flow Control (XON/XOFF)

The application can attempt to manually control the flow of data by using the following IOCtls:

- Transmit Immediate (Category 1 Function 44H)
- Stop Transmit behave as if XOFF received (Category 1 Function 47H)
- Start Transmit behave as if XON received (Category 1 Function 48H).

The device driver also will control automatically the flow of transmitted data based upon the reception of XON/XOFF characters. This is referred to as automatic transmit flow control (XON/XOFF) and is described in the section on "States of the ASYNC Device Driver" on page 9-11.

The device driver also will attempt to control the flow of data that is received by automatically transmitting XON/XOFF characters to the system it is communicating with, based on the amount of space left in the receive queue. This is referred to as automatic receive flow control (XON/XOFF) and is described in the section on "States of the ASYNC Device Driver" on page 9-11.

Line Characteristics

IOCtls can be used to control and read the baud rate, number of stop bits per character, number of data bits per character, and the parity characteristics of the line. See the section on "States of the ASYNC Device Driver" on page 9-11.

Break and Error Processing, Port Status, RI

The device driver can be commanded to transmit a Break with an IOCtl (Category 1 Function 4BH and Function 45H).

An application can detect where an error or break occurred in the input data stream by using Break Replacement Character Processing and Error Replacement Character Processing. This requires certain

binary byte combinations to be reserved for this purpose. See the section on "States of the ASYNC Device Driver" on page 9-11.

State of the COM Port

The following IOCtls can be used to determine the state of the COM port or if a given event happened. However, the exact timing relationship between this information and the specific data being received or transmitted at the time of the event is not available.

- Return COM Event Information (Category 1 Function 72H)
- Return COM Status (Category 1 Function 64H)
- Return COM Error (Category 1 Function 6DH).

Event Notification

The device driver does not provide any capabilities of event notification. For example, the only way for an application to know that RI changed state or that a Break condition occurred is to poll that status with the IOCtl Return COM Event Information. This should not be a problem for those applications that can use the automatic control modes of the device driver during the course of a communications dialog (for time-critical control functions). Polling could be adequate for non-time-critical event monitoring.

States of the ASYNC Device Driver

This section itemizes the different processing states of the ASYNC device driver, the ASYNC hardware, and the ASYNC control signals.

The items that will be discussed are:

- Baud Rate
- Data Bits
- Parity
- Stop Bits
- DTR and RTS
- DTR Control Mode
- RTS Control Mode
- Transmitting Break
- Event Word and COM Error
- COM Error
- Output handshaking using CTS, DSR, DCD
- Input sensitivity using DSR
- Automatic Transmit Flow Control (XON/XOFF)
- Automatic Receive Flow Control (XON/XOFF)
- XON/XOFF Characters
- Error Replacement Character Processing
- Error Replacement Character
- Break Replacement Character Processing
- Break Replacement Character
- Null Stripping
- Write Time-out State
- Write Time-out Value
- Read Time-out State
- Read Time-out Value
- Transmit Immediate

The following will be given for each item:

- A brief description.
- The initial (default) value.
- The device driver effect when the device driver receives an OPEN request packet for the port and the port is not already open (from a previous open without a matching close). This is called a First

Level Open. If applicable, the way the state of the device driver is affected by a close request packet.

- The MODE utility (in the OS/2 mode) used to alter the state of this item or the MODE utility (in the OS/2 mode) altering the state of this item.

Baud Rate

Baud rate is the speed for which the hardware is configured. See IOCtls "Set Baud Rate" (Category 1 Function 41H) and "Return Current Baud Rate" (Category 1 Function 61H).

Initial Value - 1200 baud.

First Level Open - No effect.

Mode Utility - User interface to change the baud rate.

Data Bits

The number of bits that are contained in each character transmitted or received by way of the communications hardware. See IOCtls "Set Line Characteristics" (Category 1 Function 42H) and "Return Line Characteristics" (Category 1 Function 62H).

Initial Value - 7 data bits.

First Level Open - No effect.

Mode Utility - User interface to change the number of data bits.

Parity

Determines whether a parity bit exists and (if appropriate) what algorithm determines its value. See IOCtls "Set Line Characteristics" (Category 1 Function 42H) and "Return Line Characteristics" (Category 1 Function 62H).

Initial Value - Even Parity.

First Level Open - No effect.

Mode Utility - User interface to change the parity characteristics.

Stop Bits

Determines the number of stop bits associated with each character transmitted or received by way of the communications hardware. See IOCtls "Set Line Characteristics" (Category 1 Function 42H) and "Return Line Characteristics" (Category 1 Function 62H).

Initial Value - 1 stop bit.

First Level Open - No effect.

Mode Utility - User interface to change the number of stop bits.

DTR & RTS

The value of the modem control signals Data Terminal Ready (DTR) and Request To Send (RTS) put out by the communications hardware. Each signal is controlled independently and can be either ON or OFF. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Return Modem Control Output Signals" (Category 1 Function 66H).

Initial Value - When the device driver starts the port during device driver initialization, their values will be turned off.

First Level Open - The signals are normally turned on but there are many conditions that may cause the signals to be affected differently. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Set Device Control Block" Information NOTE 1 (Category 1 Function 53H) for a complete explanation.

Close Considerations - A close request packet, when after processing this close request the port will not be open any more from another open without a close (last level close), will cause DTR and RTS to be turned OFF after the transmit hardware has completely transmitted all the data that it has been given to transmit by the device driver and at least 10 additional character times have elapsed.

Mode Utility - Not applicable for direct control. Indirect effects through altering processing modes of the device driver are possible.

DTR Control Mode

The different control modes for DTR are:

- Enable
- Disable
- Input Handshaking

The Enable and Disable control modes of DTR affect DTR processing during a First Level Open. When these control modes are set via the Category 1 Function 53H IOCtl, the value of the DTR signal may be modified immediately by the device driver. The action will depend on the previous control mode of DTR and the current value of the DTR modem control signal. If the control mode of DTR is Input Handshaking, then the device driver will control the DTR signal, depending on how full the receive queue is. The bits that control these states of the device driver are in the device control block. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Set Device Control Block Information" NOTE 1 (Category 1 Function 53H).

Initial Value - Enable.

First Level Open - No effect.

Mode Utility - User interface to change the DTR Control Mode of the device driver.

RTS Control Mode

The different control modes for RTS are:

- Enable
- Disable
- Input Handshaking
- Toggling on Transmit

The Enable and Disable control modes affect RTS processing during a First Level Open. When these control modes are set using the Category 1 Function 53H IOCtl, the value of the RTS signal may be immediately modified by the device driver. The action will depend on the previous control mode of RTS and the current value of the RTS modem control signal. If the control mode of RTS is Input Handshaking, the device driver will control the RTS signal, depending on how full the receive queue is. If the control mode of RTS is Toggling on Transmit then the device driver will control the RTS signal, depending on whether it transmits data. The bits that control these

states of the device driver are in the device control block. See IOCtls "Set Modem Control Signals" (Category 1 Function 46H) and "Set Device Control Block Information" NOTE 1 (Category 1 Function 53H).

Initial Value - Enable.

First Level Open - No effect.

Mode Utility - User interface to change the RTS Control Mode of the device driver.

Transmitting Break

The device driver can be transmitting a break. See IOCtls Break On (Category 1 Function 4BH) and Break Off (Category 1 Function 45H).

Initial Value - Not transmitting a break.

Close Considerations - A CLOSE request packet, when after processing this close request the port will not be open any more from another open without a close (last level close), will cause the device driver to tell the hardware not to transmit a break any more.

Mode Utility - Not applicable.

COM Event Word and COM Error Word

These two words have bits which show status of the COM port. When an event happens the appropriate bits are turned on. The bits are cleared when the word is read with the appropriate IOCtl. See IOCtl Return COM Event Information (Category 1 Function 72H) and Return COM Error (Category 1 Function 6DH).

Initial Value - All defined bits are 0.

First Level Open - All defined bits are 0.

Mode Utility - Not applicable.

Output Handshaking using CTS, DSR, DCD

This mode of the device driver can be controlled independently for each modem control signal. When this mode of the device driver is enabled, the device driver will not give data to the transmit hardware if the appropriate modem control signal is OFF. See IOCtl "Set Device Control Block" Information NOTE 3 (Category 1 Function 53H).

Initial Value - Output handshaking using CTS and DSR is enabled. Output handshaking using DCD is disabled.

First Level Open - No effect.

Mode Utility - User interface to enable/disable this mode of the device driver for CTS and DSR (independently). Mode will always disable this mode of operation of the device driver for DCD.

Input Sensitivity Using DSR

When the device driver is enabled for this mode of operation and DSR is OFF, the device driver will discard receive data. See IOCtl "Set Device Control Block Information" NOTE 4 (Category 1 Function 53H).

Initial Value - Input Sensitivity using DSR is enabled.

First Level Open - No effect.

Mode Utility - User interface to enable/disable this mode of the device driver.

Automatic Transmit Flow Control (XON/XOFF)

When the device driver is enabled for this mode of operation, the device driver will stop sending data to the transmit hardware when an XOFF is received, and resume sending data to the transmit hardware when an XON is received. See IOCtl "Set Device Control Block Information" NOTE 2 (Category 1 Function 53H).

Initial Value - Automatic transmit flow control is disabled.

First Level Open - No effect on whether the device driver is enabled or disabled for this mode of operation. The state of the device driver will be reset to show that it has not received an XOFF so it can transmit (due to automatic transmit flow control) if it is enabled for this mode of operation.

Mode Utility - User interface to enable/disable this mode of the device driver.

Automatic Receive Flow Control (XON/XOFF)

When the device driver is enabled for this mode of operation, the device driver will transmit an XOFF when its receive queue gets close to full, and an XON when its receive queue is about half full. See IOCtl "Set Device Control Block" Information NOTE 2 (Category 1 Function 53H).

Initial Value - Automatic Receive Flow Control is disabled.

First Level Open - No effect on whether the device driver is enabled or disabled for this mode of operation. The state of the device driver will be reset to show that the last flow control character automatically transmitted was an XON if it is enabled for this mode of operation.

Close Considerations - If the last automatically transmitted character by the device driver was an XOFF and a CLOSE request packet is received, (when after processing this close request the port will not be open any more from another open without a close - Last Level Close), the device driver will automatically transmit an XON, if possible.

Mode Utility - Always disables Automatic Receive Flow Control.

XON/XOFF characters

The characters used for automatic transmit and automatic receive flow control. See IOCtl "Set Device Control Block Information" NOTE 2 (Category 1 Function 53H).

Initial Value - XON is 11H and XOFF is 13H.

First Level Open - The XON and XOFF characters are reset to their initial values.

Mode Utility - No effect.

Error Replacement Character Processing

The processing that the device driver performs when a received character had an error (parity, framing, overrun, or lack of receive queue space) is determined by whether error replacement character processing is enabled (active). See IOCtl "Set Device Control Block Information" NOTE 5 (Category 1 Function 53H).

Initial Value - Error replacement character processing is disabled.

First Level Open - Error replacement character processing is disabled.

Mode Utility - No effect.

Error Replacement Character

The character value that the device driver uses if Error Replacement Character Processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 5 (Category 1 Function 53H).

Initial Value - 00H.

First Level Open - Reset to 00H.

Mode Utility - No effect.

Break Replacement Character Processing

If break replacement character processing is enabled, and the device driver detects a break condition, it will place the break replacement character in the device driver receive queue. If break replacement character processing is disabled, the device driver will not place any character in the device driver receive queue when it detects a break condition. See IOCtl "Set Device Control Block Information" NOTE 7 (Category 1 Function 53H).

Initial Value - Break replacement character processing is disabled.

First Level Open - Break replacement character processing is disabled.

Mode Utility - No effect.

Break Replacement Character

The character value that the device driver uses if Break Replacement Character Processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 7 (Category 1 Function 53H).

Initial Value - 00H.

First Level Open - Reset to 00H.

Mode Utility - No effect.

Null Stripping

If the device driver is enabled for null stripping, characters read in from the receive hardware (non error or non break) with a value of 00H are thrown away. These null characters are stripped (not checked for Automatic Transmit Flow Control) even if the XON or XOFF character has been set to 00H, and are not placed in the device driver receive queue. See IOCtl "Set Device Control Block Information" NOTE 6 (Category 1 Function 53H).

Initial Value - Null stripping is disabled.

First Level Open - Null stripping is disabled.

Mode Utility - No effect.

Write Time-out State

When the device driver processes a WRITE request packet, it can be with normal or infinite time-out processing. With normal time-out processing, if no data is given to the transmit hardware within a specified amount of time, the request will be completed. With infinite time-out processing, the request will be completed only when all the data from the request has been given to the transmit hardware. See IOCtl "Set Device Control Block" Information NOTE 8 (Category 1 Function 53H).

Initial Value - Normal Write time-out processing.

First Level Open - No effect on write time-out processing.

Mode Utility - User interface to set infinite or normal write time-out processing.

Write Time-out Value

The user specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the write time-out processing, if normal write time-out processing is enabled. See IOCtl "Set Device Control Block Information" NOTE 8 (Category 1 Function 53H).

Initial Value - 1 Minute.

First Level Open - Set to 1 Minute.

Mode Utility - No effect.

Read Time-out State

When the device driver processes a READ request packet, it can be with normal, with NO-WAIT, or with Wait For Something time-out processing. With normal time-out processing, if no data is received in the specified period of time, the request will be completed. With NO-WAIT time-out processing, the request will obtain whatever data is available in the device driver receive queue (at the time the request is processed by the device driver) and return. With Wait-For-Something time-out processing, the request will act like NO-WAIT time-out processing. However, if no data is available when the device driver processes the request, the device driver will wait until some data is available or until the request times out due to normal time-out processing. See IOCtl "Set Device Control Block Information" NOTE 9 (Category 1 Function 53H).

Initial Value - Normal Read time-out processing.

First Level Open - The device driver is set to Normal Read Time-out processing.

Mode Utility - No effect.

Read Time-out Value

The user specific value, in .01 seconds units (based on 0, where 0 = .01 seconds), is used for the read time-out processing, if normal read time-out or Wait For Something time-out processing is enabled. See IOCtl "Set Device Control Block" Information NOTE 9 (Category 1 Function 53H).

Initial Value - 1 Minute.

First Level Open - Set to 1 Minute.

Mode Utility - No effect.

Transmit Immediate

The device driver may be told to transmit a byte immediately, bypassing the normal file system write requests (bypassing the data to be transmitted in the transmit queue). Only one character at a time can be waiting to be transmitted "immediately." See IOCtl "Transmit This Byte Immediately" Category 1 Function 44H.

Initial Value - There is no character waiting to be transmitted immediately.

First Level Open - There is no character waiting to be transmitted immediately.

Close Considerations - A CLOSE request packet, when after processing this close request the port will not be open any more (from another open without a close), will cause the device driver to attempt to transmit the character waiting to be transmitted immediately. If the device driver cannot transmit the character waiting to be transmitted immediately (See IOCtl "Transmit this Byte Immediately" Category 1 Function 44H), then it will not try to transmit the character and proceed with the close processing.

Mode Utility - Not applicable.

Reserved Device Names- COM1-N

The device name AUX does not appear in the Device Header of the ASYNC device driver. The ASYNC device driver does not support the reserved name AUX for either DOS mode applications or OS/2 mode applications.

Personal Computer AT Considerations- COM1, COM2

The Personal Computer AT ASYNC device driver will have device names COM1 and COM2 in its Device Header.

Device name COM1 will correspond to the ASYNC hardware with its base address in 40:0 (during initialization) and device name COM2 will correspond to the ASYNC hardware with its base address in 40:2 (during initialization). After the ASYNC device driver initialization, the contents of the 40: area are not monitored by the ASYNC device driver. The values in 40: are set to 0 for any COM port that is initialized. If these values are restored later in the same power on session, the integrity of the ASYNC device driver could be adversely affected by any process that directly accesses hardware through the 40: area.

It should be noted that if the adapter card is configured as a secondary COM port, the system may not always treat it as COM2.

Note: The mapping of the 40: area to COMn must be consistent across all device drivers that support the ASYNC hardware in the Personal Computer AT hardware environment.

Refer to the SETCOM40 Utility for more information.

PS/2 Considerations- COM1 - 3

The PS/2 ASYNC device driver will have device names COM1, COM2, and COM3 in its Device Header.

Device name COM1 will correspond to the first LID in the Advanced BIOS common data area with the ASYNC device ID. Device name COM2 will correspond to the second LID and device name COM3 will correspond to the third LID in the Advanced BIOS common data area with the ASYNC device ID. The Advanced BIOS architecture ensures that the ordering in the 40: area will match the ordering of the LIDs in the common data area. Compatibility BIOS supports up to 4 ASYNC devices on PS/2 and the device driver assumes that the order of the Logical IDs match the order of the addresses of these devices in the 40: area.

Additional ASYNC devices may be supported by additional device drivers.

The mapping of the ASYNC Logical ID ordering to COMn must be consistent across all device drivers that support the ASYNC hardware in the PS/2 hardware environment.

Initialization / Resource Management

The device driver is loaded and initialized with a DEVICE= statement in CONFIG.SYS. The name of the device driver for the Personal Computer AT is COM01.SYS and the name of the device driver for the PS/2 is COM02.SYS. The device driver does not process any parameters on the DEVICE= statement. It is the responsibility of the installation process or the user to have the correct DEVICE= statement in the CONFIG.SYS file, depending on whether the OS/2 is installed on:

- IBM Personal Computer AT and IBM Personal Computer XT Model 286 - COM01.SYS
- IBM Personal System/2™ Models 50, 60, and 80 - COM02.SYS

During initialization, the device driver will attempt to free memory from its data segment for ports it does not need to support. The device driver will not remove device names from its Device Header for ports that do not get initialized.

The device driver will not deinstall a device if the system requests it. If another device driver wishes to support a port already supported by this device driver, it needs to initialize before this ASYNC device driver. Taking the appropriate action, means this device driver will not prevent the other device driver from supporting the COM port in question. There are many ways this can be done, depending on the system. See initialization considerations below.

Dynamic ownership of a given COM port between different device drivers in the same boot of OS/2 is not supported. Only one device driver can support a given COM port in a given boot of OS/2.

If a supported port encounters a condition where it fails to initialize, a message will be displayed to indicate the nature of the condition.

If installation fails due to errors for all supported ports, the system will display a message and pause, allowing corrective measures to be taken.

Personal Computer AT Initialization Considerations

The device driver will not attempt to initialize or support a port if it does not get the INIT request packet for the port's corresponding device name.

If the device driver gets the INIT request packet for a given device name it will check to see if a valid I/O address (2F8H or 3F8H) is in the appropriate 40: area (that corresponds to that device name). COM1 is 40:0 and COM2 is 40:2.

If the 40: area does not have a valid I/O address, the device driver fails the initialization of this port and will not support this port.

Otherwise, the device driver attempts to get exclusively the interrupt level that corresponds to the I/O address for this port. If the interrupt level is not available, then the device driver fails the initialization of this port and will not support this port.

If the interrupt level is available, then the device driver will give back the interrupt level, set to 0 the 40: area address for this port (this will normally prevent INT 14H from functioning for this port), initialize the port, and set up to support the port during this start-up of OS/2.

In summary, in order for the device driver to support a port on the Personal Computer AT, the following must be true:

- The device driver must get an INIT request packet for the device name.
- The 40: area that corresponds to the device name must have a valid I/O address.
- The appropriate interrupt level must be available for exclusive use, even though the device driver will not claim the interrupt level for exclusive use during initialization.

The device driver claims ownership of the port by not DEINSTALLING the corresponding device name and by setting to 0 the corresponding 40: area.

Another device driver can cause this device driver not to claim a port by initializing before this device driver and doing at least one of the following:

- Not allowing this device driver to receive an INIT request packet for a given device name.
- Putting an invalid I/O address in the corresponding 40: area (for example, 0).
- Owning exclusively the appropriate interrupt level at initialization time.

PS/2 Initialization Considerations

The device driver will not attempt to initialize or support a port if it does not get the INIT request packet for the port's corresponding device name.

If the device driver gets the INIT request packet for a given device name then it will attempt to claim ownership of the specific LID position for the ASYNC device ID that corresponds to the device name being initialized.

If the LID is not available, then the device driver fails the initialization of this port and will not support this port.

If the LID is available, then the device driver will initialize the port and set up support for this port during this start-up of OS/2. The

device driver will set to 0 the appropriate 40: area (this will normally not allow INT 14H to function for this port).

In summary, for the device driver to support a port on PS/2, the following must be true:

- The device driver must get an INIT request packet for the device name.
- The ASYNC LID corresponding to the device name must be available.

The device driver claims ownership of the port by not DEINSTALLING the corresponding device name and by claiming the appropriate ASYNC logical ID. The device driver also sets to 0 the corresponding 40: area.

Another device driver can cause this device driver not to claim a port by initializing before this device driver and doing at least one of the following:

- Not allowing this device driver to receive an INIT request packet for a given device name.
- Claiming the appropriate ASYNC LID.

Access Authorization

The device driver does not prevent multiple processes from concurrently opening the same device name. This is the responsibility of the user or the subsystems that reside between the applications and the device driver. Allowing multiple processes to use (OPEN) concurrently the same device name could cause unexpected results.

Data Translation / Monitor Support /Spooler Support

The device driver provides no data translation, code page, or monitor support. It is the responsibility of the application or subsystem to provide any function required in these areas.

Spooling from LPT to COM is supported by the print spooler but spooling from COM to LPT or COM to COM is not supported. The level of code page support provided by the print spooler is available through LPT and the print spooler.

Any requests for registering or opening a monitor chain to COMn will be rejected by the device driver.

The device driver deals with binary data and provides no special processing in the area of "binary" or "ASCII" mode.

File System Requests

Open Processing

The device driver does not claim the interrupt level the port is on until the port is open. If the interrupt level is not available, the Open request packet is failed. The interrupt level is claimed exclusively on the Personal Computer AT. The interrupt level is claimed shareable on PS/2.

If a timer tick handler is not available during First Level Open processing, the open request may fail.

If the device driver receives an OPEN request packet and the COM device is not already open (from a previous open without a close), the device driver does special processing. See "States of the ASYNC Device Driver". This is called a First Level Open. If a subsequent open request is issued before a previous First Level Open request has completed, the device driver may process the open request packets in a different order than they were issued. This could cause the First Level Open to take effect at a different time from what the application was expecting.

An OPEN request should never be issued until a previous last level CLOSE request has completed. If an OPEN request is issued before a previous last level CLOSE has completed, the function performed by a last level close and a First Level Open may not occur.

On the Personal Computer AT, if the port is not already open (First Level Open), the device driver will attempt to clear out any data in the receive hardware. On PS/2, if the port is not already open (First Level Open), the device driver will rely on the "reset / initialize" Advanced BIOS function to reset and clear the UART receive hardware.

Close Processing

An application should never close an open handle to a COM port while there are requests outstanding to that handle. If a request has not completed, it may be waiting for time-out processing. IOCtls may be used to determine the current time-out processing and to change the current time-out processing.

If the device driver receives a CLOSE request packet, the device driver will do some special processing. When after processing this close request the port will not be open any more from another open without a close - Last Level Close. See "States of the ASYNC Device Driver" on page 9-11. The device driver, (when processing a close request that causes the port not to be open any more - Last Level Close), will:

- Clear the receive and transmit queues.
- Turn break off if currently transmitting a break.
- Clear any character waiting to be transmitted immediately if it cannot be transmitted. If it can be transmitted, the device driver will make sure that it is given to the transmit hardware.
- If currently enabled for automatic receive flow control (XON/XOFF) and the last character that the device driver automatically transmitted was an XOFF, the device driver will attempt to automatically transmit an XON, if possible.
- Wait until all the data in the transmit hardware has been physically transmitted.
- Unclaim the interrupt level.
- Turn DTR & RTS OFF if they are not already OFF. The device driver will wait the specified number of character times first (see "States of the RS232 Device Driver").

Read Processing

The device driver will begin processing Read requests in the order that they are received by the device driver.

Note: This may not be the same order that the requests were issued by the application. If the device driver receives more than one Read request, the request packet will be queued on the Read request packet queue for later processing.

Applications may not see read requests completed in the order that they were received by the device driver. The order of the data placed in the read requests will reflect the order the requests were received by the device driver.

The data for the read requests comes from the device driver receive queue. Because of time-out processing, it is normal for the total number of read characters requested not to be read. This is not considered an error. The request is completed when time out is completed or when the amount of data requested is placed in the read buffer.

The different kinds of Read Time-out processing are discussed in the "States of the ASYNC Device Driver" on page 9-11.

To reduce the probability of a device driver receive queue buffer overrun, the communications protocol should take into account the size of the device driver receive queue.

Write Processing

The device driver will begin processing write requests in the order that they are received by the device driver.

Note: This may not be the same order that the requests were issued by the application. If the device driver receives more than one write request, the request packet will be queued on the Write request packet queue for later processing.

Applications may not see write requests complete in the order that they were received by the device driver. The order of the data transmitted due to the Write requests will reflect the order that the requests were received by the device driver.

The data from the Write requests are placed in the device driver transmit queue. The number of characters written are considered to be the number of characters given to the transmit hardware and not the number of characters placed in the device driver transmit queue. Because of time-out processing, it is possible that the total number of Write characters requested will not be transmitted. This is not considered an error. The request is completed when completes or when the amount of data requested is given to the transmit hardware (but not actually transmitted at the physical RS232C interface).

The different kinds of write time-out processing are discussed in the section on "States of the ASYNC Device Driver" on page 9-11.

If infinite write time-out processing is enabled, then it is the responsibility of the application to monitor the status of the write requests. The application may have to issue an IOCtl to disable infinite write time-out processing to cause the write request to complete (without all the data being transmitted). If an application does not wish to check that all the data is given to the transmit hardware on each write request, the application can use the infinite time-out processing mode of the device driver to ensure that all the data has been given to the transmit hardware before the request completes.

In order to increase the throughput (ratio of number of characters transmitted per second to the baud rate), the application should keep the write requests as large as possible.

DOS Mode Considerations / Restrictions

DOS mode applications that go directly to the COM hardware or that use INT 14H are not supported. The 40: area is set to 0 by the device driver for those ports serviced by the device driver.

Previous-level DOS device drivers that go directly to COMn hardware are not supported.

The DOS 3.3 CTTY Utility is not supported.

The AUX device name does not appear in the device header for the ASYNC device driver. AUX does not correspond to COM1.

We strongly recommend applications using a serial printer from the DOS mode spool print data to the print spooler through the appropriate LPT handles.

The only support to COMn in the DOS mode is through the file system. Family API restrictions should be consulted for the subset of Category 1 IOCtls that are supported in the DOS mode.

COM support in the DOS mode is incompatible with COM support in DOS. COM support in DOS 3.3 is half duplex through INT 14H. COM support in OS/2 has the following attributes:

- Full duplex
- Interrupt driven
- Sophisticated time-out processing
- Many different control modes of the modem control signals
- Logical flow control capabilities
- No application knowledge of state of modem control signals and data flow on a character basis.

The device driver makes no attempt to restrict or mold the function of file system requests because they may have come from the DOS mode. To achieve the full capabilities of the file system access to COM, the application needs access to the full range of Category 1 IOCtls. The design and externals of the ASYNC device driver are based on the requirements of new OS/2 mode applications that use the RS232-C port of the system.

If the DOS mode is concurrently sharing a COM port with OS/2 mode (not recommended), then care must be taken not to switch away from the DOS mode while the DOS mode application has requests outstanding in the file system. Switching away from the DOS mode, while file system requests are outstanding, could cause the device driver not to process certain file system requests that are outstanding (for a given port).

Refer to the SETCOM40 command in the IBM Operating System/2™ *User Reference*.

Performance

The achievable performance is very sensitive to its environment. The type and amount of other system activity will determine the achievable performance. On PS/2 the number of COM ports or other devices on the same interrupt level will significantly affect the achievable performance level.

Trying to receive data at too high a baud rate could cause hardware overrun errors or receive queue overrun errors. Receive queue overrun errors are easily solved by adjusting the communications protocol to the size of the device driver receive queue.

Trying to transmit data at too high a baud rate also could cause the performance of the OS/2 system to be lessened severely.

Configuration

The baud rate can be set with the MODE command or with an IOCtl. The baud rate should not be set to values which may cause receive overruns or adverse OS/2 system performance effects.

Pointer Draw (Screen) Device Driver

The Mouse Pointer Draw (Screen) device driver draws the system default and user-supplied mouse pointer images on the screen at interrupt time.

In the OS/2 mode, the pointer draw device driver (POINTDD.SYS) provides two levels of display mode support:

- Full draw support.
- Disabled state support.

Full Draw Support

Full draw support includes the full pointer drawing capabilities of a pointer device driver. The following display modes are supported at this level:

- Text only

  modes 0, 1, 2, 3 and their + and * variations.
  mode 7 and its + variation.

These modes are always accepted by the CheckModeProtect function.

If a mode other than one of the above is set using VioSetMode and pointer drawing is not disabled, the pointer device driver will report an error to the mouse device driver causing the mouse device driver to shut down. When the Mouse device driver is shut down, the API is active but no interrupt data can be returned. In this case, a status flag is set and is available using MouGetDevStatus.

Disabled State Support

Disabled State support is available when the pointer draw functions have been disabled by the MouSetDevStatus function. When this occurs, an extended set of display modes is supported by the CheckModeProtect function. These display modes are:

- Text modes

  modes 0, 1, 2, 3 and their + and * variations.
  mode 7 and its + variation.

- Graphics modes

  modes 4, 5, 6, 7, D, E, F, 10, 11, 12, and 13.

- The IBM Personal System/2™ Display Adapter advanced function modes.

In the disabled state, no pointer drawing is performed by the pointer draw device driver. Instead, the application is expected to perform the pointer draw functions.

While in this state, subsequent MouSetDevStatus function calls to re-enable the pointer drawing will continue to be bypassed until VioSetMode resets the screen mode to one of the supported modes.

In the DOS mode, the pointer draw device driver (POINTDD.SYS) provides full pointer draw functions for the following modes:

- Text modes

  modes 0, 1, 2, 3, and their + and * variations.
  mode 7 and its + variation.

- Graphics modes

  modes 4, 5, 6. D, E, F, and 10.

All other modes will cause the mouse device driver to shut down.

Mouse Device Driver

Mouse Device Overview
This section describes the standard mouse device driver interface provided by OS/2 for both DOS mode and OS/2 mode applications.

Mouse Devices Supported: Mouse device drivers are supported in both DOS mode and OS/2 mode for the following devices:

- Microsoft®[1] Bus (parallel) Mouse for IBM Personal Computers (part number 037-099, 100ppi)
- Microsoft® Bus (parallel) Mouse for IBM Personal Computers (part number 037-199, 200ppi)
- Microsoft® Mouse (serial) for IBM Personal Computers (part number 039-099, 100ppi)
- Microsoft® Mouse (serial) for IBM Personal Computers (part number 039-199, 200ppi)
- PC Mouse™[2] (serial) by Mouse Systems (part number 900120-214, 100ppi)
- Visi On™[3] Mouse (serial) (part number 69910-1011, 100ppi)
- Microsoft® Mouse (InPort) for IBM Personal Computers (part number 037-299, 200ppi)
- PS/2 Mouse (In-Processor) for IBM Personal System/2™ Computers (part number 6450350, 200ppi)

While the OS/2 mouse device drivers support both DOS mode and OS/2 mode applications, the means by which DOS mode applications access the mouse device differs from OS/2 mode application access.

DOS mode applications must use the interrupt 33H (INT 33H) interface described later in this chapter. OS/2 mode applications must use the

---

1 Microsoft is a registered trademark of Microsoft Corporation

2 PC Mouse is a trademark of Metagraphics/Mouse Systems

3 Visi On is a trademark of Visi On Corporation

OS/2 mode MOU*xxx* API also described in this chapter and in *Technical Reference, Vol. 2.*

OS/2 mode applications may not use the INT 33H API nor may DOS mode applications use the Mou*Xxx* mouse interface.

PS/2 Mouse: The PS/2 mouse is an "in-processor" mouse. It is not compatible with existing mouse devices. The PS/2 mouse requires its new device driver. The PS/2 mouse is supported for both the DOS mode and OS/2 mode applications but only on a PS/2 processor. The PS/2 mouse is not supported on the Personal Computer AT or Personal Computer XT Model 286.

The Personal Computer AT serial mouse attachment cards will not fit in the PS/2 chassis. However, the OS/2 mouse devices are supported on PS/2 by allowing the user to connect the mouse devices to the PS/2 serial port. The PS/2 mouse device driver implementation utilizes ABIOS Pointing Device ABIOS commands.

OS/2 D/D Serial Mouse Interrupt Sharing: OS/2 Personal Computer AT Serial mice do not share interrupts. Mouse device drivers attempt to capture the COM line specified by the SERIAL keyword on the mouse device's DEVICE statement in the CONFIG.SYS file. If exclusive use is denied, mouse device driver installation will fail. The OS/2 PS/2 mouse device drivers are modified to allow interrupts to be shared with other ASYNC drivers. The request for interrupts is non-exclusive.

On the PS/2, the mouse device drivers will always receive exclusive access to the requested I/O port. However, the mouse device driver will never receive exclusive access to the device interrupt.

System Install ensures that mouse device driver initialization takes place prior to ASYNC device driver initialization. This allows the ASYNC device driver to determine that it is not responsible for servicing that port. This will ensure that mouse device drivers will not be preempted from the COMx ports by the ASYNC device drivers.

For PS/2, OS/2 Mouse device drivers utilize the ABIOS command "Return LID Parameters" to determine which interrupt level they are executing on. Under no circumstances can the Mouse Device drivers select a predetermined interrupt.

Mouse Screen Resolutions: The screen resolution is determined by either system default or by the application's issuing an explicit VioSetMode call in the OS/2 mode or an INT 10H, AH = 0 in the DOS mode or INT 10H, AH = 11 Character Font Generator Selections. When in DOS mode the virtual display resolution is used. The virtual display resolution depends on the display mode selected. The following display modes are supported:

Mode	Type	Text Resolution	Graphics Resolution	Virtual (X,Y) Coordinates	Cell Size
0	BW Text	40 x 25	320 x 200	640 x 200	8 x 8
0+	BW Text	40 x 25	320 x 350	640 x 200	8 x 14
0*	BW Text	40 x 25	360 x 400	640 x 200	9 x 16
1	CO Text	40 x 25	320 x 200	640 x 200	8 x 8
1+	CO Text	40 x 25	320 x 350	640 x 200	8 x 14
1*	CO Text	40 x 25	360 x 400	640 x 200	9 x 16
2	BW Text	80 x 25	640 x 200	640 x 200	8 x 8
2+	BW Text	80 x 25	640 x 350	640 x 200	8 x 14
2*	BW Text	80 x 25	720 x 400	640 x 200	9 x 16
3	CO Text	80 x 25	640 x 200	640 x 200	8 x 8
3+	CO Text	80 x 25	640 x 350	640 x 200	8 x 14
3*	CO Text	80 x 25	720 x 400	640 x 200	9 x 16
4	Graphics	-	320 x 200	640 x 200	2 x 1
5	Graphics	-	320 x 200	640 x 200	2 x 1
6	Graphics	-	640 x 200	640 x 200	1 x 1
7	Mono	80 x 25	720 x 350	640 x 200	9 x 14
7+	Mono	80 x 25	720 x 400	640 x 200	9 x 16
D	Graphics	-	320 x 200	640 x 200	2 x 1
E	Graphics	-	640 x 200	640 x 200	1 x 1
F	Graphics	-	640 x 350	640 x 350	1 x 1
10	Graphics	-	640 x 350	640 x 350	1 x 1

Mouse Installation: Mouse support is installed at IPL (start-up) time. The mouse support may be tailored according to the user's needs. This is accomplished via use of the CONFIG.SYS file to define system mouse requirements.

The following describes the CONFIG.SYS DEVICE= keywords available to customize the mouse subsystem and related mouse device driver installation.

- The SERIAL= keyword is used to specify the communications port that a serial mouse device is connected to.

Note: Utilities support up to eight COM ports on PS/2 (COM1 - COM8) and up to two on Personal Computer AT (COM1 and COM2). Device drivers support up to three on PS/2 (COM1 - COM3) and up to two on Personal Computer AT (COM1 and COM2). COM1, COM2 and COM3 should be used for ASYNC.

This keyword is not valid for nonserial mice, (Microsoft Bus Mouse, Microsoft InPort Mouse, and IBM PS/2 Mouse, for example).

If this keyword is not present (for serial mice), the default used is COM1. Otherwise, either COM1 or COM2 must be specified for the Personal Computer AT or PS/2. COM3 through COM8 may be specified for serial mice on a PS/2 only.

- The QSIZE= keyword is used to specify the event queue length to be used for all OS/2 mode sessions.

  If this keyword is not present, a default of 10 (maximum queue elements) is used. If a queue length is specified, the keyword must be followed by a signed integer in the range:

  1 < = integer < = 100

  Each queue element occupies 10 bytes. Therefore, the default event queue size allocates 100 bytes of event queue buffer space per session.

  A maximum of 16 sessions is allowed to utilize mouse support at any one time. This is true even if the number of sessions specified for the system is greater than 16. If an attempt is made to open mouse support for more than 16 sessions, an error will occur.

- The MODE= keyword, enables the user to specify whether the mouse support is required for DOS mode only, OS/2 mode only or both modes.

  The acceptable MODE= values are:

  B = Both DOS mode and OS/2 mode support
  P = OS/2 mode support only
  R = DOS mode support only

  The default for the MODE= option is Both. Consequently, if this MODE= option is not specified, both DOS mode and OS/2 mode device driver support will be loaded into the system.

For example, if the user specifies a CONFIG.SYS file with a DEVICE = mouseA00.sys and none of the other parameters, the internal result would be as if the user had specified the following statement:

```
DEVICE=mouseA00.sys,SERIAL=COM1,MODE=b,QSIZE=10
```

Mouse Device Driver Packaging: The DOS mode and OS/2 mode device drivers are contained within the same executable modules. Each of the supported devices is supplied with a named device driver containing both DOS mode and OS/2 mode function.

Personal Computer AT Mouse Device Drivers: The following drivers support the Personal Computer AT and Personal Computer XT Model 286. (These are not to be used for PS/2 hardware.)

- MOUSEA00.SYS = PC Mouse by Mouse Systems - Serial (part number 900120-214)
- MOUSEA01.SYS = Visi On Mouse - Serial (part number 69910-1011)
- MOUSEA02.SYS = Microsoft Mouse for IBM Personal Computers - Serial (part numbers 039-099 and 039-199)
- MOUSEA03.SYS = Microsoft Mouse for IBM Personal Computers - Parallel (part numbers 037-099 and 037-199)
- MOUSEA04.SYS = Microsoft Mouse for IBM Personal Computers - InPort (part number 037-299)

PS/2 Mouse Device Drivers: The following drivers support PS/2 hardware:

- MOUSEB00.SYS = PC Mouse by Mouse Systems - Serial (part number 900120-214)
- MOUSEB01.SYS = Visi On Mouse - Serial (part number 69910-1011)
- MOUSEB02.SYS = Microsoft Mouse for IBM Personal Computers - Serial (part numbers 039-099 and 039-199)
- MOUSEB05.SYS = PS/2 In-Processor Mouse (part number 6450350)

The system loads the entire module into storage during initialization. The mouse device driver examines the parameter on the MODE= keyword (if one exists) on the CONFIG.SYS, DEVICE=MOUSE*xx*.SYS line specifying the mouse device driver to determine whether both modes of support are required.

If OS/2 mode is not required, the storage occupied by the OS/2 mode-only portions of the mouse device driver support is removed from storage.

Mouse Pointer Draw Implementation: Communication between both DOS mode and OS/2 mode mouse device drivers and the pointer draw screen device routine is conducted by way of a FAR call from the mouse driver to the entry point of the screen pointer draw routine.

The setup required for the mouse device driver prior to issuing the call to the pointer draw routine is as follows:

- Set DS:SI to point to the session data control block described below.
- Set the screen_func field to indicated 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 session data control block.

When the mouse device driver calls the screen pointer draw routine, the draw routine must issue a CLI to disable interrupts. The mouse device driver will have disabled the IOCtls and the mouse device interrupts. These will be enabled by the mouse device driver after the pointer draw routine returns to it.

All addresses to data obtained via a DevHlp_AllocPhys is 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 OS/2 mode and segment:offset for DOS mode) with the DevHlp_PhysToVirt call. The resulting selector is temporary and will be valid 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
- Reenable interrupts.

It is possible for the screen pointer draw routine to do repetitively the following in order to bypass the 400 microsecond time limit:

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

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

The functions to be supported by the screen pointer draw routine are:

- screen_func = 0 = DrawPointer (Bimodal)
- screen_func = 1 = RemovePointer (Bimodal)
- screen_func = 2 = FreePointerMemory (Bimodal)
- screen_func = 3 = CheckModeProtect (OS/2 mode only)
- screen_func = 4 = CheckModeReal (DOS mode only)
- screen_func = 5 = GetPointerMemory (Bimodal)

The following chart outlines the interrupt state of the 8259 for the commands that may be executed with calls from the mouse device driver to the screen pointer draw routine. The chart also describes the execution modes from which the commands may be invoked.

Function Type	**Interrupt Status**	**Call Modes**
DrawPointer	Disabled @ 8259 Level	Interrupt, User, Kernel
RemovePointer	Disabled @ 8259 Level	Interrupt, User, Kernel
FreePointerMemory	Disabled @ 8259 Level	User, Kernel
CheckModeProtect	Disabled @ 8259 Level	Kernel
CheckModeReal	Disabled @ 8259 Level	User
GetPointerMemory	Disabled @ 8259 Level	User, Kernel

Details concerning each of these commands follow:

- DrawPointer draws the current mouse pointer image if it is not within the area defined by the collision area definition fields and if it is not already visible on the screen.

- RemovePointer removes the current mouse pointer image from the screen if it was visible. Before the call, an advisory parameter (CX) is set as follows:

  CX = 0 The RemovePointer call may not be immediately followed by a DrawPointer call. The Pointer device driver must consider such a call an unconditional request for pointer removal, and honor the remove pointer.

  CX = 1 The RemovePointer call will be immediately followed by a DrawPointer call (that is, the pointer image is to be moved rather than removed). The Pointer device driver can consider this RemovePointer call as advisory.

  A Pointer device driver which examines the CX value may decide, if CX is 1, to defer removal of the pointer image until the DrawPointer call can be examined also. Depending on the sophistication of the pointer driver, this can provide more efficient image updating. However, the pointer driver may choose to ignore the advisory indication (CX = 1), in which case it must always remove the pointer image.

  For the case where CX is 1, it is guaranteed that for the given screen group, no other function call will be made to the pointer driver between the RemovePointer and the DrawPointer calls. However, there is no guarantee that a function call for another screen group will not occur. Thus, the pointer driver making use of the advisory nature of RemovePointer must save pointer information on a per-screen-group basis. The Screen_Tble field can be used by the pointer driver to extend the per-screen-group information as needed.

- FreePointerMemory can only be issued after a RemovePointer to free both the current pointer image buffer and its associated screen restore buffer. This call has no effect if the default pointer is the screen pointer.

- CheckModeProtect verifies the requested mode_data structure values as either supportable or unsupportable:
  - If the requested OS/2 mode screen mode is unsupportable, the mouse screen device driver will set an error code of 1 in the AX register and return.
  - If the requested mode is supportable, the function will set the control block mode data fields accordingly and set a valid return code of 0 in AX. In addition, the cell sizes (in the control block) for the new mode must be filled in.

    Mode_data is a 12 byte data structure pointed to by ES:DI. The mode_data structure is defined below.
- CheckModeReal verifies the requested mode_data structures values as either supportable or unsupportable as follows:
  - If the requested DOS mode screen mode is unsupportable, the mouse screen device driver will set an error code of 1 in the AX register and return
  - If the requested mode is supportable, the mouse screen device driver will set the control block mode data fields and set a valid return code of 0 in AX. In addition, the cell sizes (in the control block) for the new mode are filled in.

    Real_mode_data is a 3-byte data structure pointed to by ES:DI. The Real_mode_data structure is defined below.
- GetPointerMemory will only be issued after a FreePointerMemory in order that the mouse screen device driver may 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 ES:DI. The pointer definition control block contains two addresses. They are:
  - The address of the pointer image buffer
  - The address of the pointer definition record

  The pointer definition record is defined below.

  If the pointer image data is incomplete, unsupportable, or 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 OK, this function:

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

In order to maximize pointer image draw performance, there are restrictions on defining graphics pointer images. These limitations follow:

- Graphics modes utilizing the 320x200 resolution require pointer images be defined with 4 pixels per byte.
- Graphics modes utilizing the 640x200 resolution require pointer images be defined with 8 pixels per byte.

In other words, graphics pointer images must be defined in byte-width multiples. Non-byte width definitions will be accepted by the pointer draw routine but may result in unexpected pointer images appearing on the display screen.

The pointer image does not need to be drawn on the screen by the pointer draw routine. It is feasible for the pointer draw routine to dispatch a process at level 2 or 3 and have that process affect pointer draw operations.

This approach may be of particular use to those subsystems and Environment Managers which conduct a large number of screen or processing functions for each interrupt.

Mouse Device Driver - Default Pointers: The default pointer images supplied by the system are a default text pointer for OS/2 mode only and a default text and graphics pointer for DOS mode. Two images are supplied; Default text image, and 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 as an upward pointing arrow leaning toward the right side of the screen.

The same graphics pointer image is used for both medium (320x200) and high resolution (640x200) 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 follow:

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

DefGrph Struct  ;default graphics pointer

ANDmask DB 11111111B,11000011B ;default graphics ptr
        DB 11111111B,10000011B ;AND mask
        DB 11111111B,00000011B
        DB 11111110B,00000011B
        DB 11111100B,00000011B
        DB 11111000B,00000011B
        DB 11110000B,00000011B
        DB 11100000B,00000011B
        DB 11000000B,00000011B
        DB 10000000B,00000011B
        DB 10000000B,00000011B
        DB 11110000B,00000011B
        DB 11110000B,00000011B
        DB 11100000B,00111111B
        DB 11100000B,01111111B
        DB 11100000B,01111111B
XORmask DB 00000000B,00000000B ;XOR mask
        DB 00000000B,00010000B
        DB 00000000B,00110000B
        DB 00000000B,01110000B
        DB 00000000B,11110000B
        DB 00000001B,11110000B
        DB 00000011B,11110000B
        DB 00000111B,11110000B
        DB 00001111B,11110000B
        DB 00011111B,11110000B
        DB 00000001B,11110000B
        DB 00000011B,00010000B
        DB 00000011B,00000000B
        DB 00000110B,00000000B
        DB 00000110B,00000000B
        DB 00000000B,00000000B

DefGrph Ends
```

For all system-supported modes, TotLength is equal to:

- 4 for text modes.
- 64 for graphic modes.

Mouse Device Driver - Control Blocks: Internal mouse pointer control blocks are described below. These structures are used by both the mouse device DOS mode and OS/2 mode drivers and 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 Session Data Area Template (118 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
;
;   Session control data sub-table  (next 40 Bytes)
;
  Rowscale_Fact DW  ? ; M   row coordinate scale factor
  Colscale_Fact DW  ? ; 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   # 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_PID        DW  ? ; M   PID blocked on event queue
  Eq_Size       DB  ? ; M   # of used elements in queue
  Chain_Size    DB  ? ; M   # 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 14 Bytes)
;
  MB_Len        DW  ? ; MON Monitor Buffer Length (14 bytes)
  MFlags        DW  ? ; MON monitor flags
  EMask         DW  ? ; MON event occurrence mask value
  Time          DD  ? ; MON event time stamp (Time of Day in milliseconds
  Row_Pos       DW  ? ; MON current pointer row coordinates
  Col_Pos       DW  ? ; MON current pointer column coordinates
;
;   Display information data sub-table (next 64 Bytes)
;             Display Mode fields
;
  Length        DW  ? ; BP  len of display mode fields (bytes)
  Mtype         DB  ? ; BP  mono text/color text/color graphic
  Color         DB  ? ; BP  # of color bits (graphic 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_Size DW  ? ; BP  graphics col res/ text col res
  Row_Cell_Size DW  ? ; BP  graphics row res/ text row res
;
;    Mouse Pointer fields
;
  Ptr_Flags     DW  ? ; P   pointer image visible/hidden
  Ptr_Height    DW  ? ; P   height of ptr image (resolution units)
  Ptr_Width     DW  ? ; P   width of ptr image (resolution units)

  Ptr_Row_Pos   DW  ? ; BM  current ptr row coord position
  Ptr_Col_Pos   DW  ? ; BM  current ptr col coord position

  Ptr_Row_Ref   DW  ? ; P   row coord ptr shape reference pxl
  Ptr_Col_Ref   DW  ? ; P   col coord ptr shape reference pxl
  Ptr_Image_Buf DD  ? ; P   physical addr to ptr image bufr
  Ptr_Buf_Len   DW  ? ; P   pointer image buffer len (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_Savstartodd DW ?; P   pointer save start odd pointer
  Ptr_Savendodd DW  ? ; 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 coord position
  Area_Col_Pos  DW  ? ; BM  area starting col coord position
  Area_Row_End  DW  ? ; BM  area ending row coord position
  Area_Col_End  DW  ? ; BM  area ending col coord 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  ptr shape buffer byte length
   width        DW  ? ; BM  pointer width shape dimension
   height       DW  ? ; BM  pointer height shape dimension
   col_hot      DW  ? ; BM  ptr col coord hot spot pixel
   row_hot      DW  ? ; BM  ptr row coord hot spot pixel
ptr_template ENDS

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

;               Mode Data Record Template
;          (12 byte structure for OS/2 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 dd to pointer draw dd on MouSetPtrShape only.
;
; ES : DI points to this control block
; -------

ptr_def_cb  STRUC
     addr1      DD  ? ; BM  addr to ptr definition record
     addr2      DD  ? ; BM  addr to ptr image buffer
ptr_def_cb  ENDS

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

;                 CheckModeReal Data Structure
;
; The structure's first (3) fields (R_Mode, Ex_Rows,
; Ex_Points), are copied from the DOS mode BIOS data
; area.  Indicates that mode to which the DOS mode is
; about to be set by an INT 10H, AH=0 (setmode) call, or is
; about to be changed by INT 10H, AH=11, AL=1x (character
; generator calls).  The Pointer Draw Device Driver will
; return to the Mouse Device Driver the new Display Mode's
; virtual coordinate maximum grid values via the virt_rows and
; virt_cols structure fields.
;
; ES : DI points to this control block
; ------------------------------------

Real_Mode_Data STRUC  ;     For real mode support
     r_mode     DB  ? ; BM  Standard Disp Mode, CGA
     ex_rows    DB  ? ; BM  Number of Rows, EGA
     ex_points  DW  ? ; BM  Standard Disp Mode, EGA
     virt_rows  DW  ? ; BP  New Disp Mode Virt Coord Row Max
     virt_cols  DW  ? ; BP  New Disp Mode Virt Coord Col Max
Real_Mode_Data ENDS
```

OS/2 Mode Mouse Support
This section describes the standard OS/2 mouse device support for OS/2 mode applications.

Overview: Mouse device drivers have characteristics which are quite different from most other devices. They are read-only devices which provide data at approximately 30 events per second. The data is structured, that is, it will arrive as a packet of data containing absolute screen location, button up/down and other information.

The OS/2 mode mouse driver model described in this section is designed to provide a basic, machine-independent, high-performance interface. Applications and Environment Managers may use this interface to obtain mouse device services.

The Base Mouse Subsystem (BMS) is a dynamic link module which 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. One handler per session is allowed.

Normally, the system supplied default handler is the session's mouse handler. However, Environment Managers, OEM, 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 sessions, provided it uses MouRegister with each of those sessions.

Pointer image updating is executed by the mouse device driver utilizing functions supplied by the display device drivers. The pointer updating occurs at interrupt time, 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 have been well separated from those of a screen driver. Pointer updating functions call the display device drivers rather than attempting to draw the pointer image directly. This maintains independence between the mouse and display devices. It does require that custom display device

drivers conform to the pointer device driver interface to allow the pointer shape to be drawn and also requires that the application synchronize display access.

Pointer Draw Installation: The mouse pointer image update design allows the update routine to be installed with the video subsystem. Custom and OEM video subsystems, which allow display modes not supported by the base video subsystem, may implement interrupt-time screen pointer image updating by providing screen image draw routines for execution by the pointer device driver.

Screen pointer draw routines are called by the mouse pointer device driver at interrupt time. The screen pointer draw routines are installed as character device drivers at start time.

The necessary steps for installing a screen pointer image draw routine are outlined below:

- Pointer draw routines are installed at IPL time by including them on a DEVICE= keyword in the CONFIG.SYS file as named character device drivers. No special mechanism is needed. The default pointer draw device driver file named POINTDD.SYS is needed.
- The display (screen) =IOCtl (category 3, 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 MouOpen to a mouse handle, the mouse subsystem handler/router will inspect the stack to determine if the call specified a non-system pointer draw device driver name.
  - If the pointer is 0, the mouse subsystem will use the default (system supplied) device driver. This means that the session is restricted to display modes 0 through 7.
  - If the pointer was not 0, the mouse subsystem will follow the pointer to get the ASCIIZ name of the pointer draw device driver.
- The mouse subsystem handler/router will OPEN the pointer draw character device driver using DosOpen. Using the device handle returned by the OPEN, the handler/router will then issue the category 3, Function 72H screen IOCtl to get the entry address

(selector : offset) of the pointer draw device driver. The mouse device driver will issue FAR calls to this entry address when the mouse has a pointer manipulation request.

- The mouse subsystem handler/router will then do a DosOpen to the mouse device driver. The return value from this call will be a standard OS/2 device handle.
- The mouse subsystem handler/router will use 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 7, Function 5AH mouse IOCtl addressed to the mouse device driver's DOS handle.
- The mouse device driver will call the pointer draw routine with each request without their being linked prior to start time. Linkage will be established (via the IOCtls) for each pointer draw device driver specified by a MouOpen.
- There may be only one pointer draw routine (driver) for each session.
- This mechanism applies to OS/2 mode only.

Handler/Router: There are three aspects to an OS/2 mouse handler/router:

- Mou*Xxx* API to allow applications to avoid the specifics of the low level IOCtl interface
- Circular buffers to receive events
- Pointer management interface.

The driver is installed as a character device, with the name "MOUSE*xxx*.SYS." The MouOpen function call initializes the device (sets initial coordinates, checks for mouse presence).

The Mou*Xxx* interface allows the caller to obtain information about the current state of the mouse, set parameters, allow the application to determine which events are to be passed into the device circular buffer, and others.

The circular I/O buffer is a high-efficiency buffer shared by all client applications in the session and the mouse device driver. There is only one queue per session, no matter how many applications within

the session are utilizing the mouse device. The driver uses this interface to provide "events." The caller can specify what constitutes an event. Examples of events are pressing buttons and moving a mouse.

Events are time-stamped so that a higher-level interface library package can provide such features as pressing the mouse button twice.

Coordinates: Coordinates are mouse event's "absolute" position relative to the top left corner (0,0) of the display screen. This means that the units in which the mouse position is reported depends on the display mode in which the session is executing. There are two different possibilities:

- In TEXT mode, pointer position is reported in CHARACTER units.
- In GRAPHICS mode, pointer position is reported in PIXELS.

By supplying pointer coordinates as offsets to absolute screen position, higher level library support for translating data into an absolute coordinates may no longer be necessary.

For those systems that wish to operate in terms of "relative" mouse movement (mickey) displacements, the MouSetDevStatus call allows the library support or the application to set the mouse device driver to return mickey movements and not screen coordinates.

Motion: The unit of motion for a mouse is known as a "mickey." This is similar to the pixel, the unit of addressability on a screen.

The OS/2 mouse driver provides calls to determine the number of units of motion per centimeter, so that an application, window manager or other package can relate motion to a physical screen.

Mou *Xxx* and IOCtl Calls: The mouse IOCtl is category 7. Applications should not concern themselves with the details of the mouse IOCtl interface. Instead, the 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.

All mouse device driver IOCtl functions have an application-level MOU API equivalent function. The IOCtl function codes and their MOU API equivalents are as follows:

IOCtl Fcn	MOU API Function	Function Performed
IOMR_NB	MouGetNumButtons	Get # of mouse buttons
IOMR_MC	MouGetNumMickeys	Get # of mickeys/centimeter
IOMR_GS	MouGetDevStatus	Get device status flags
IOMW_DS	MouSetDevStatus	Set device status flags
IOMR_QS	MouGetNumQueEl	Get event queue status
IOMR_RD	MouReadEventQue	Read event queue contents
IOMR_GF	MouGetScaleFact	Get current scaling factors
IOMW_SS	MouSetScaleFact	Set new scaling factors
IOMR_GM	MouGetEventMask	Get current event mask
IOMW_EM	MouSetEventMask	Set new event mask
N/A	MouOpen	Open mouse support
N/A	MouClose	Close mouse support
N/A	MouRegister	Install a mouse subsystem
N/A	MouDeRegister	Deinstall a mouse subsystem
N/A	MouInitReal	Initialize DOS mode driver
IOMW_SP	MouSetPtrShape	Assign new pointer shape
IOMW_GP	MouGetPtrShape	Assign new pointer shape
IOMW_DP	MouDrawPtr	Unmark collision area
IOMW_RP	MouRemovePtr	Mark collision area

OS/2 Mode Mouse API

Please refer to *Technical Reference, Vol. 2* for a discussion of the function calls dealing with Mouse OS/2 mode APIs. A summary of Mouse OS/2 mode API descriptions follows:

MouRegister	Register mouse subsystem
MouDeRegister	Deregister mouse subsystem
MouInitReal	Initialize DOS mode pointer draw
MouOpen	Opens the mouse device for the current session
MouClose	Closes the mouse device for the current session.
MouDrawPtr	Release screen area for device driver use
MouRemovePtr	Reserve screen area for application use
MouFlushQue	Flush mouse event queue
MouGetDevStatus	Query current pointing device driver status flags

MouGetEventMask	Query current pointing device one-word event mask
MouGetNumButtons	Query number of buttons
MouGetNumMickeys	Query number of mickeys per centimeter
MouGetNumQueEl	Query current status for the pointing device event queue
MouGetPtrPos	Query current pointer position
MouGetPtrShape	Query pointer shape and size
MouGetScaleFact	Query scale factors for the current pointing device
MouReadEventQue	Read the pointing device event queue
MouSetDevStatus	Set device status flags
MouSetEventMask	Assign new event mask to the current pointing device
MouSetPtrPos	Set current pointer position
MouSetPtrShape	Set pointer shape and size
MouSetScaleFact	Set scale factors for the current pointing device
MouSynch	Synchronize (serialize) access to the mouse device driver

The MouOpen and MouClose commands are mapped by the Mouse Base Subsystem Router into the DosOpen and DosClose IOCtl commands, respectively. MouRegister and MouDeRegister are directed to the higher level Mouse Base Subsystem Router and do not translate down to the IOCtl device driver level.

The Base Mouse Subsystem receives all Mou*Xxx* function calls. Function specific data is passed on the user's stack in the following format:

```
Mouse Router return address    2 words <== Top of Stack
Value of application DS reg.   1 word
Entry point return address     1 word
Function code                  1 word
Application (caller) ret addr  2 words
Function specific parameters   2 - 10 bytes depending on call
```

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

Mouse events are placed in a circular I/O buffer. The conditions which generate an event are controllable through the event mask

feature and are available by way of the MouSetEventMask call or the equivalent mouse IOCtl command.

Mouse events have the following format:

```
WORD    Event Mask
DWORD   time stamp in milliseconds
WORD    Row absolute / mickeys
WORD    Column absolute / mickeys
```

These fields have the following meaning:

Event Mask: This indicates which event(s) are in this record. See the MouGetEventMask call description in *Technical Reference, Vol. 2* for details on the event mask bit definitions.

Time: This is a time stamp for the event. It is provided so that higher level interface packages can provide features like two mouse button presses with selective timing, and so that mouse and keyboard monitor events can be synchronized. The time value is the number of milliseconds since the last IPL.

Row and Column / Mickeys: This may indicate either absolute position of the mouse pointer shaper relative to the top left corner of the display screen or mickey mouse movement.

If reporting coordinates (the default), they will be in either pixel or character offsets, depending on whether the display mode for the session is graphics or text, respectively.

The application must explicitly invoke reporting of mouse mickey units with the MouSetDevStatus call. In this case, events are reported in units of mouse movement. For rows, a negative number means movement toward the upper part of the screen. For columns, a negative number means movement toward the left part of the screen.

Pointer: Maintenance of the pointer (shape and location) is performed by the mouse device driver. An application must provide the mouse driver with a pointer image that the driver will use to draw the pointer for that session.

Applications utilizing mouse services need to 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 API provides three commands to accomplish these functions:

- MouSetPtrShape
- MouDrawPtr
- MouRemovePtr

It is the responsibility of the application to synchronize pointer operations between itself and the pointer shape draw routine.

Display Modes Supported: Graphic modes are only supported in the OS/2 mode if the application performs all drawing and moving of the pointer. The controlling application must first issue MouSetDevStatus and indicate no interrupt time pointer drawing, plus pointer movement reported in mickeys. Next, issue VioSetMode to the desired graphics modes. The graphic modes that are supported are 4, 5, 6, 0DH, 0EH, 010H, 011H, 012H, 013H and the 8514/A adapter Advanced Functions modes.

DOS Mode Mouse Support

This section describes the standard OS/2 mouse device support for DOS mode applications.

Overview: DOS mode mouse support is based on the Microsoft INT 33H support.

DOS mode Mouse support preserves the INT 33H interface and is substantially compatible with existing Microsoft support.

Pointer Draw Installation: DOS mode pointer draw installation is implemented via the MouInitReal call. The shell issues the MouInitReal call during shell initialization. The sequence of events is as follows:

At SYSINIT (IPL) time, the shell (System Session Manager) issues MouInitReal.

In order to establish addressability between the DOS mode mouse device driver and the DOS mode screen pointer draw routine:

- The shell issues a MouInitReal to open the system default Pointer Draw Device Driver.
- The MouInitReal issues category 3 (screen) IOCtl 72H to the DOS mode/OS/2 mode (shared) mode screen pointer draw device driver. This IOCtl will return the address of the pointer draw routine entry point.
- The MouInitReal will then issue category 7 (mouse) IOCtl 5BH to the DOS mode mouse device driver. This IOCtl passes the pointer draw address obtained from the preceding category 3, function 72H IOCtl to the mouse device driver.
- MouInitReal will then return to the shell with a completion code indicating the result of the DOS mode mouse initialization process.

Handler/Router: The Mouse Handler/Router is applicable only to OS/2 mode mouse support. In OS/2 mode, it directs operations among multiple sessions. In addition, it allows the Mou*Xxx* calls to be intercepted and synchronized between multiple processes in a session. This interception is done by an environment Manager, mouse subsystem, or a sophisticated application.

DOS mode mouse support calls may be intercepted by hooking the INT 33H vector. While there are multiple OS/2 mode sessions allowed, there may not be more than one DOS mode session. Consequently, there is no need for, and no support provided for, a DOS mode mouse handler/router.

Coordinates: The OS/2 mode mouse reports its coordinate position in absolute displacement (characters or pixels) from the upper left corner of the screen. In a similar manner, DOS mode mouse support reports mouse coordinates relative to the upper left corner of the screen.

In contrast, the DOS mode mouse support reports the position in virtual screen units. The virtual display coordinates are relative to the display mode. Refer to the table on page 9-36 for a list of the supported display modes and their virtual display coordinates.

The OS/2 Mouse Device Drivers do not limit the virtual display coordinate settings. However, if an application wishes to define the display with a larger virtual coordinate grid than the physical, it should also be prepared to perform the pointer image drawing because the OS/2 virtual coordinate support will always map back to the virtual display space. The relative displacement of one unit will depend on the dimensions of the screen and the associated resolution for the mode setting on the display while the DOS mode is the foreground session.

Motion: As with OS/2 mode mouse support, the unit of motion for a mouse is known as a "mickey". This is similar to the pixel, the unit of addressability on a screen.

IOCtl Calls: There is only one IOCtl supported for the DOS mode mouse device driver.

Portions of the mouse device driver are used by both the DOS mode and OS/2 mode device drivers. This shared portion supports the Category 7 (mouse), Function 5BH IOCtl for the DOS mode. This IOCtl is only used by the shell and then only at shell initialization time.

The OS/2 mode shell (System Session Manager) always exists in the OS/2 system. It is the shell (at shell initialization time) that calls for DOS mode mouse device initialization. The purpose of this shell call is to determine the entry point of the screen pointer draw device driver. It is this entry point which the mouse device driver will call on interrupts to have the pointer image updated for the DOS mode session.

There are no other mouse IOCtls in OS/2 that are supported by the DOS mode device driver.

Events: The Microsoft INT 33H DOS mode mouse API is designed to detect and report "changes" in the state of the mouse. Consequently, the mouse support reports events such as:

- Button Press data
- Button Release data
- Button Number
- Mouse movement

Pointer: The DOS mode mouse API supports application pointer image setting. Two commands are provided to enable the application to modify the DOS mode pointer shape. They are:

- Set Text Pointer Shape
- Set Graphics Pointer Shape

The first command is used while the display is in text modes. The second command is used while the display is in graphics modes.

Display Modes Supported: All text modes are supported (0, 1, 2, 3 and 7). Graphic modes 4, 5, 6, 0DH, 0EH, 0FH and 010H are also supported.

Mouse Monitors

Some applications need to view mouse device events as they arrive from the device driver. These applications may wish to consume some of the mouse events, or they may wish to replace some mouse events with one or more other mouse events. This is made possible by the "mouse monitor" function.

The mouse device driver supports device monitors. The device driver passes information to the monitor in packets consisting of a word of monitor flags plus the standard mouse device driver event buffer. The packet format is as described below:

```
WORD  <00> -- Monitor Flags -- open, close, etc.
WORD  <02> -- Event mask (see MouGetEventMask for definitions)
DWORD <04> -- Time stamp in milliseconds
WORD  <08> -- Absolute horizontal (x or row) screen position
WORD  <0A> -- Absolute vertical (y or column) screen position
```

The following DevHlp monitor control functions are used by the OS/2 Mouse Device drivers to implement mouse monitors:

- MonitorCreate
- Register
- Deregister
- MonWrite
- MonFlush

The OS/2 Mouse Device Driver uses the DevHlp monitor control functions for the following situations:

MonitorCreate

Creates a monitor chain for each OS/2 session, supported by the system, when the Mouse Device Driver receives and processes the system's INIT request.

Removes each support session's monitor chain when a DEINSTALL request is received and processed.

Register

Add a monitor to a monitor chain for the current session when a register request is received and processed.

The size of the mouse device driver's data buffer is 16 bytes. This is the value to be used in calculating the sizes of the input/output buffers required for the DosMonReg call.

Deregister

Remove a monitor from the current session's monitor chain when a deregister request is received and processed.

MonWrite

Provide the current session's monitor chain event data at interrupt time.

MonFlush

Flush all data from session's monitor chain.

For additional information concerning device monitor function, see "Device Monitor Services" on page 6-32.

DOS Mode INT 33H Mouse API

Please refer to *Technical Reference, Vol. 2* for a discussion of the function calls dealing with Mouse OS/2 mode APIs. OS/2 supports a subset of the Microsoft DOS INT 33H mouse API. Refer to the table on page 9-36 for the display modes supported.

The Microsoft INT 33H mouse API is available to only those applications executing in the DOS mode. OS/2 mode applications must use the MOU*xxx* mouse device interface.

The Mouse Device Driver provides DOS mode applications with an INT 33H interface to the pointing device hardware. The DOS mode support is not equivalent to the OS/2 OS/2 mode support instead, it preserves the Microsoft INT 33H mouse interface.

The DOS mode Mouse does not support:

- Device handles
- Monitor chains
- IOCtl direct function access.
- MOU API function calls.

All DOS mode mouse functions are accessed on the software INT 33H interface. When a software interrupt 33H is detected, a DOS mode Mouse Handler Routine is invoked. All function relevant information is supplied by the caller in the following seven registers:

- AX
- BX
- CX
- DX
- SI
- ES
- DI

There are two states for the DOS mode Mouse support:

- The OS/2 Mouse Device Driver is loaded into the operating system. When this state is active, all DOS mode Mouse functions listed in this section are available for user support.
- The OS/2 Mouse Device Driver processes a DeInstallation request. In this state the DOS mode Mouse support is limited to

INT 33H, Function call 0. This is also true if an invalid/unsupported display mode was set.

Mouse Device Driver Interfaces/Requirements: To get DOS mode mouse support, use MODE=R or MODE=B on the DEVICE= statement of CONFIG.SYS.

Mouse Device Driver Button Definitions: The DOS mode button definition depends on the number of buttons on the mouse device.

Button definitions are used on functions 3, 5, 6, and 12. Button number assignments are as follows:

Two-button mouse:

Bit#	***Mouse Button***
1	Rightmost button
0	Leftmost button

Three-button mouse

Bit#	***Mouse Button***
2	Center button
1	Rightmost button
0	Leftmost button

Mouse Device Driver Function Summary: The software INT 33H interface is provided only for DOS mode support. The INT 33H Mouse API provides the following functions:

#	***Function Performed***
0	Mouse Installed Flag and Reset
1	Show Mouse Pointer
2	Hide Mouse Pointer
3	Get Mouse Pointer Position & Button Status
4	Set Mouse Pointer Position
5	Get Button Press Information
6	Get Button Release Information
7	Set Min and Max Horizontal Position
8	Set Min and Max Vertical Position
9	Set Graphics Pointer Shape
10	Set Text Pointer Shape
11	Read Mouse Motion Counters
12	Set User-Defined Subroutine Input Mask
13	Light Pen Emulation Mode ON
14	Light Pen Emulation Mode OFF
15	Set Mickey/Pixel Ratio
16	Conditional OFF
19	Set Double Speed Threshold
20	Swap User-defined Subroutine
21	Query Save Mouse State Storage Requirements
22	Save Mouse Driver State
23	Restore Mouse Driver State

The function number and the function specific parameters are passed to the Mouse Device Driver in the four general purpose registers and the SI, DI and ES registers.

- The AX register is always used to contain the requested function number.
- The BX, CX, and DX registers are used as needed for function-specific input parameters.
- SI and DI are used for function call 16.
- ES is used for function calls 9 and 12.

On return from a software interrupt 33H function call, the general purpose registers contain return codes and/or mouse requested data items. The registers used are function-specific and detailed under each individual call description.

All input parameters are checked on function calls requiring parameters.

If an INT 33H function call is issued when the Mouse Hardware/Software is not properly initialized, the call will return an error code of 0 in AX.

The INT 33H Interface is based on the concept of a virtual display screen coordinates. All coordinates are input/output relative to a default range of rows and columns. The default virtual range is required for the OS/2 Mouse Device Driver to perform the pointer image tracking on the display. However, an application is not stopped from altering the virtual display coordinate limits through the INT 33H interface. If an application alters virtual display coordinate grids to be greater than the physical display resolution, the pointer tracking will be unpredictable and all virtual coordinates read from the INT 33H interface may be inaccurate. This is properly supportable if the application which set the virtual coordinate ranges also assumes responsibility of pointer image drawing. The application can assume the INT 33H motion counters will not be affected and therefore is a stable source of pointer image motion data.

The mouse device driver maps the physical display resolution to the virtual display screen coordinate system independent of the physical display mode.

INT 33H-0 Installed Flag and Reset

Purpose Determines if the DOS mode Mouse device is present.

If the appropriate mouse device hardware and software are available, this function will return a value of -1 in the AX register. In addition, this call will set/reset all of the software tracking mechanisms used by the other DOS mode mouse function calls to their default values.

If the mouse hardware and/or software is not available for interaction, or if the video mode is not supported, this call will return a value of 0 in AX.

Input Parameters:

AX = function code of 0
BX = not used
CX = not used
DX = not used

Return codes/data values:

If mouse support is not available, then
  AX = 0
  If mouse support is installed then
    BX = -1
else (mouse support is available)
  AX = -1
  BX = number of mouse device buttons supported.

The following table defines the default function values set when Mouse support is available (AX = -1):

Function	*Default value*
Mouse pointer position	Screen center
Pointer redraw flag	-1, Pointer hidden
Graphics pointer image	Mouse default image
Text pointer image	Reverse video
Interrupt call mask	All 0's, no routine in use
Light pen emulation mode	Enabled
Mickey/pixel ratio (horizontal)	8 to 8
Mickey/pixel ratio (vertical)	16 to 8
Min/Max ptr position (horizontal)	Display/mode dependent
Min/Max ptr position (vertical)	Display/mode dependent

INT 33H-1 Show Pointer

Purpose Requests that the mouse device driver draw the pointer image and update the pointer redraw flag.

This function determines if the mouse pointer image is already visible (pointer redraw flag = 0). If the pointer image is already visible, this function is a no-op.

If the pointer redraw flag is not 0, this call increments the pointer redraw flag. After the pointer redraw flag is incremented, it is checked again to see if it is equal to 0.

If the redraw flag was incremented to 0, the pointer image is redrawn on the display screen.

If the internal cursor flag is already 0, this function has no effect.

This function also clears conditional off (function 16) values.

Input Parameters:

AX = function code of 1
BX = not used
CX = not used
DX = not used

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-2 Hide Pointer

Purpose Requests the mouse device driver to hide the pointer image and update the pointer redraw flag.

This function decrements the mouse pointer redraw flag. If the pointer image is already hidden, the call returns to the application. If the mouse pointer redraw flag was decremented to -1, then the pointer image is hidden.

When the cursor is hidden, the mouse device driver continues to track the motion of the pointer on the screen. It simply does not draw the pointer image on the display.

This function is used to ensure that the pointer device driver will not interfere with data being written on the screen by application programs.

This function always decrements the cursor flag regardless of its current value.

Input Parameters:

AX = function code of 2
BX = not used
CX = not used
DX = not used

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-3 Get Position & Button Status

Purpose Returns the current mouse button status and pointer image screen coordinates ((columns, rows) or (x, y)).

Input Parameters:

AX = function code of 3
BX = not used
CX = not used
DX = not used

Return codes/data values:

AX = Unchanged
BX = mouse button status
CX = horizontal (x or column) pointer coordinate
DX = vertical (y or row) pointer coordinate

The mouse button status value is a bit mask indicating which mouse button(s) are currently pressed/released.

A SET bit is defined as a button being pressed.

Bit#	***Value/meaning***
3-15	Reserved = 0

For two button mouse

Bit#	***Value/meaning***
2	Reserved = 0
1	Set if rightmost button pressed
0	Set if leftmost button pressed

For three button mouse

Bit#	***Value/meaning***
2	Set if center button pressed
1	Set if rightmost button pressed
0	Set if leftmost button pressed

The pointer coordinates are relative to the range defined for the mouse under the virtual terminal concept. The ranges are display mode dependent. See "Mouse Screen Resolutions" on page 9-36 for a description of the virtual screen resolution by mode.

INT 33H-4 Set Pointer Position

Purpose Assigns the mouse pointer image to a new screen location.

If the coordinates are too large, or too small, the mouse pointer position is set to the maximum or minimum screen values, respectively.

Input Parameters:

AX = function code of 4
BX = not used
CX = new pointer horizontal coordinate
DX = new pointer vertical coordinate

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-5 Get Button Press Information

Purpose Returns a specified button's status information.

Button status information consists of:

- Up/down state of all buttons
- A button press counter value
- The last button press screen position coordinates

The button press counter contains the number of times the requested button was pressed since the last time this call was issued.

The button press counter is reset to 0 by this call. There is no overflow checking performed for the button press counter(s) when they are updated.

The button press screen position coordinates are always reported in a virtual display mode value.

The pointer coordinates are relative to the range defined for the mouse under the virtual terminal concept. The ranges are display mode dependent. See "Mouse Screen Resolutions" on page 9-36 for a description of the virtual screen resolution by mode. The values follow:

Input Parameters:

AX = function code of 5
BX = button status requested
0 = leftmost button
1 = rightmost button
2 = center button
CX = not used
DX = not used

Return codes/data values:

AX = Bit-mapped as follows:

Bit#	*Meaning*
0	= 0 If leftmost button is UP, 1 if down
1	= 0 If rightmost button is UP, 1 if down
2	= 0 If center button is UP, 1 if down
3-15	= Not used

BX = Button counter value
CX = Last button press horizontal coordinate position
DX = Last button press vertical coordinate position

This function uses AX to return a value. If the input parameter in BX is illegal, then the output registers are returned as:

AX = 0
BX = 0
CX = 0
DX = 0

INT 33H-6 Get Button Release Information

Purpose Returns a specified button's status information.

Button status information consists of:

- up/down state of all buttons
- a button release counter value
- the last button release screen position coordinates.

The button release counter will contain the number of times the requested button was released since the last time this call was issued.

The button release screen position coordinates are always reported in a virtual display mode value.

The pointer coordinates are relative to the range defined for the mouse under the virtual terminal concept. The ranges are display mode dependent. See "Mouse Screen Resolutions" on page 9-36 for a description of the virtual screen resolution by mode. The returned values are:

Input Parameters:

AX = function code of 6
BX = button status requested
  0 = leftmost button
  1 = rightmost button
  2 = center button
CX = not used
DX = not used

Return codes/data values:

AX = Bit-mapped as follows:

Bit#	***Meaning***
0	= 0 If leftmost button is UP, 1 if down
1	= 0 If rightmost button is UP, 1 if down
2	= 0 If center button is UP, 1 if down
3-15	= Not used

BX = Button counter value
CX = Last button press horizontal coordinate position
DX = Last button press vertical coordinate position

This function uses AX to return a value. If the input parameter in BX is illegal, then the input registers are returned as:

AX = 0
BX = 0
CX = 0
DX = 0

INT 33H-7 Set Min & Max Horiz Position

Purpose Assigns virtual screen minimum and maximum horizontal coordinate positions.

By defining virtual screen horizontal minimum and maximum coordinates, the pointer image is limited to a subset of the physical display horizontal movement area.

If the value is too small, the current minimum value is used. Values larger than the maximum, the physical display resolution, may be used but with unpredictable pointer image tracking results.

If the maximum value is less than the minimum, then the two values are swapped.

If the pointer image is outside of the area when the call is made, it is moved to just inside the area.

Input Parameters:

AX = function code of 7
BX = not used
CX = minimum virtual screen horizontal position
DX = maximum virtual screen horizontal position

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-8 Set Min & Max Vert Position

Purpose Assigns virtual screen minimum and maximum vertical coordinate positions.

By defining virtual screen vertical minimum and maximum coordinates, the pointer image is limited to a subset of the physical display vertical movement area.

If the value is too small, the current minimum value is used. Values larger than the maximum, the physical display resolution, may be used but with unpredictable pointer image tracking results.

If the maximum value is less than the minimum the two values are swapped.

If the pointer image is outside of the area when the call is made, it is moved to just inside the area.

Input Parameters:

AX = function code of 8
BX = not used
CX = minimum virtual screen vertical position
DX = maximum virtual screen vertical position

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-9 Set Graphic Pointer Block

Purpose Assigns a new graphics mouse pointer image.

This function defines the shape, color and hot spot of the mouse pointer when the display is in graphics mode.

The following pointer image information must be provided:

- New pointer image horizontal hot spot coordinate
- New pointer image vertical hot spot coordinate
- Pointer to the new pointer image buffer

The hot spot coordinates are pixel value indices relative to the upper left corner of the pointer. The relative hot spot coordinates must be in the range of +/- 16.

The pointer image buffer must have a length of 64 bytes. The pointer image buffer is logically divided into two bit level masks:

- The first 32 bytes define the screen mask. The screen mask determines whether the pointer pixels are part of the shape or background.
- the last 32 bytes define the pointer mask. The pointer mask determines how the pixels under the pointer contribute to the color of the pointer.

The pointer draw routine first logically ANDs the screen mask with the 256 bits of data that define pixels under the pointer. Then it logically XORs the pointer mask with the result of the AND operation.

In modes 6, ODH, OEH, OFH and 010H each screen bit defines the color of a single pixel. Thus, one bit in the screen mask and one bit in the pointer mask define the pixel's color when the pointer is over it.

In modes 4 and 5, each pair of screen bits defines the color of a single pixel. Consequently, a pair of bits in the screen mask and a pair in the pointer mask define a pixel's color.

Input Parameters:

AX = function code of 9
BX = pointer hot spot (horizontal position)
CX = pointer hot spot (vertical position)

DX = address of screen and pointer masks
ES = segment of screen and pointer masks

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

INT 33H-10 Set Text Pointer

Purpose Defines a text (character) pointer image.

Input Parameters:

AX = Function code of 10
BX = pointer select
0 selects the software text pointer
1 selects the hardware cursor
CX = Screen mask value/hardware cursor start scan line
DX = Pointer mask value/hardware cursor stop scan line

For the software text pointer, the masks (CX and DX) are bitmapped as follows:

Bit#	***Meaning***
15	Blinking
14-12	Background Color
11	Intensity
10-8	Foreground Color
7-0	Character

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-11 Read Mouse Motion Counters

Purpose Returns the number of mickeys the mouse has moved horizontally and vertically since the last time this function was called.

The returned value is between -32768 and 32767.

A positive number indicates motion to the right for horizontal motion, and to the bottom for vertical motion.

This call sets the counts to 0. Overflow is ignored.

Input Parameters:

AX = Function code of 11
BX = Not used
CX = Not used
DX = Not used

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Horizontal count
DX = Vertical count

INT 33H-12 Set User-defined Subroutine

Purpose Sets the call mask and subroutine address for the mouse hardware interrupts.

The mouse driver will call the designated subroutine if any of the mask conditions are met.

To cause the subroutine to be invoked for a certain condition, set the corresponding bit in the call mask to a 1. If the subroutine is not to be invoked for a condition, the corresponding bit should be a 0.

Input Parameters:

AX = Function code of 12
BX = Not used
CX = Call mask
DX = Offset of subroutine
ES = Segment of subroutine

The call mask is a word value with the following bit map:

Bit#	*Meaning*
15-7	Reserved (0)
6	Center button released
5	Center button pressed
4	Rightmost button released
3	Rightmost button pressed
2	Leftmost button released
1	Leftmost button pressed
0	Pointer position changed

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

When the mouse driver calls the subroutine, it loads the following values into the general purpose registers:

AX = Condition mask (similar to the call mask except a bit is set only if the condition has occurred.)
BX = Button State
CX = Pointer Coordinate (horizontal)

DX = Pointer Coordinate (vertical)
SI = Last raw vertical mickey count read from mouse
DI = Last raw horizontal mickey count read from mouse

Note: Because the DS register contains the mouse driver data segment, the user's subroutine must set it to its own data segment value.

INT 33H-13 Light Pen Emulation On

Purpose Instructs the mouse device driver to emulate a light pen.

Calls to the PEN function in IBM Basic will return the pointer position at the last "pen down".

The "pen down" state is created by pressing the leftmost and rightmost buttons at the same time.

The "pen off the screen" state occurs when either button is up.

Input Parameters:

AX = Function code of 13
BX = Not used
CX = Not used
DX = Not used

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-14 Light Pen Emulation Off

Purpose Disables the mouse device driver light pen emulation.

Input Parameters:

AX = Function code of 14
BX = Not used
CX = Not used
DX = Not used

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-15 Set Mickey/Pixel Ratio

Purpose Sets the mickey-to-pixel ratio for horizontal and vertical mouse motion.

The ratios specify the number of mickeys per eight pixels. The values must be in the range:

1 < = value < = 32767

The default horizontal ratio is 8 to 8. With this ratio, mouse travel to move the pointer image completely across the screen horizontally depends on the mouse device being used. The following table describes the movement by device.

The default vertical ratio is 16 mickeys to 8 pixels. With this ratio, mouse travel to move the pointer image completely across the screen vertically depends on the mouse device being used. The following table describes the movement by device:

Manufacturer	Mouse Part No	PPI	Mouse Type	Horiz Travel	Vert Travel
Microsoft	037-099	100	Bus	6.4 in	4.0 in
Microsoft	037-199	200	Bus	3.2 in	2.0 in
Microsoft	037-299	200	InPort	3.2 in	2.0 in
Microsoft	039-099	100	Serial	6.4 in	4.0 in
Microsoft	039-199	200	Serial	3.2 in	2.0 in
Mouse Systems	900120-214	100	Serial	6.4 in	4.0 in
Visi On	69910-1011	100	Serial	6.4 in	4.0 in
PS/2	6450350	200	In-Proc	3.2 in	2.0 in

Input Parameters:

AX = Function code of 15
BX = Not used
CX = Horizontal Mickey/Pixel Ratio
DX = Vertical Mickey/Pixel Ratio

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-16 Conditional Off

Purpose Defines a region on the screen for updating.

If the mouse pointer is in the defined region, or moves into it, this function will hide the defined region while it is being updated. After this function is called, function 1 must be called later to show the pointer again.

This function is similar to function 2, but is intended for advanced applications that need quicker screen updates. Because of the number of parameters required, this function cannot be called from interpreted BASIC.

Input Parameters:

AX = Function code of 16
BX = Not used
CX = Left Column (x or width) screen coordinate
DX = Upper Row (y or height) screen coordinate
SI = Right Column (x or width) screen coordinate
DI = Lower Row (y or height) screen coordinate

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-19 Set Dbl Speed Threshold

Purpose Sets the threshold speed for doubling pointer motion on the screen.

The default value is 128 mickeys per second. If the mouse moves faster than this number, pointer motion doubles in speed.

Input Parameters:

AX = Function code of 19
BX = Not used
CX = Not used
DX = Threshold speed in mickeys/second

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged

INT 33H-20 Swap User-defined Subroutine

Purpose Sets the call mask and subroutine address for the mouse hardware interrupts and returns the previous values of the call mask and subroutine address.

The mouse driver will call the designated subroutine if any of the mask conditions are met.

To cause the subroutine to be invoked for a certain condition, set the corresponding bit in the call mask to a 1. If the subroutine is not to be invoked for a condition, the corresponding bit should be a 0.

Input Parameters:

AX = Function code of 20
BX = Not used
CX = Call mask
DX = Offset of subroutine
ES = Segment of subroutine

The call mask is a word value with the following bit map:

Bit#	*Meaning*
15-7	Reserved (0)
6	Center button released
5	Center button pressed
4	Rightmost button released
3	Rightmost button pressed
2	Leftmost button released
1	Leftmost button pressed
0	Pointer position changed

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Previous call mask
DX = Offset of previous subroutine
ES = Segment of previous subroutine

When the mouse driver calls the subroutine, it loads the following values into the general purpose registers:

AX = Condition mask (Similar to the call mask except a bit is set only if the condition has occurred.)

BX = Button State

CX = Pointer Coordinate (horizontal)

DX = Pointer Coordinate (vertical)

SI = Last raw vertical mickey count read from mouse

DI = Last raw horizontal mickey count read from mouse.

Note: Because the DS register contains the mouse driver data segment, the user's subroutine must set it to its own data segment value.

Because this call occurs at interrupt time, it should process the information quickly and return. If it does not, interrupts could be lost.

INT 33H-21 Query Save Mouse State Storage Requirements

Purpose Get the size of the buffer required to store the current state of the mouse driver.

Input Parameters:

AX = Function code of 21
BX = Not used
CX = Not used
DX = Not used

Return codes/data values:

AX = Unchanged
BX = Size of buffer required to store the mouse state
CX = Unchanged
DX = Unchanged

INT 33H-22 Save Mouse Driver State

Purpose Save the mouse driver state in a user buffer.

This function moves the mouse driver data, required to restore the mouse driver state, into the user defined buffer. This function is used in conjunction with function 23 when the mouse driver state must be saved and later restored.

Input Parameters:

AX = Function code of 22
BX = Not used
CX = Not used
DX = Offset of buffer
ES = Segment of buffer

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

INT 33H-23 Restore Mouse Driver State

Purpose Restore the mouse driver state from a user buffer.

This function restores mouse driver data previously saved by function 22 (Save Mouse Driver State). This function is used in conjunction with function 23 when the mouse driver state must be saved and later restored.

Input Parameters:

AX = Function code of 23
BX = Not used
CX = Not used
DX = Offset of buffer
ES = Segment of buffer

Return codes/data values:

AX = Unchanged
BX = Unchanged
CX = Unchanged
DX = Unchanged
ES = Unchanged

VDisk Device Driver

OS/2 includes a Virtual Disk installable device driver. This driver, called VDISK.SYS supports the command line configuration shown below.

In CONFIG.SYS:

```
device=[d:][path]vdisk.sys [bbbb] [ssss] [dddd]
```

bbbb The first numeric value, if present, is the disk size in kilobytes. The default is 64, the minimum value is 16.

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

dddd The third numeric value, if present, is the number of root directory entries. The default is 64, the minimum value is 2 and the maximum value is 1024.

VDISK 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 is rounded up to 32, the next multiple of 16, (there are 16 32-byte directory entries in 512 bytes).

Note: In the event that there is not enough memory to create the VDISK volume, VDISK will try to make a DOS volume of 16K size.

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

Note: The OS/2 file system cannot accept a root directory containing more than 255 sectors. For example, a 64K RAM disk with 128 byte sectors and 1024 directory entries requires 256 sectors. This should be considered by the user when setting these parameters. In the above example, the maximum number of directories the user should specify when using 128 byte sectors is 1020.

CLOCK$ Device Driver

OS/2 assumes that the 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. OS/2 uses this bit to identify the device driver, and therefore, this device can take any name. The device has been named "CLOCK$" to avoid possible conflicts with any files named "CLOCK."

OS/2 on the 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 family and is therefore not programmed by the DOS mode applications. When the DOS mode is in the foreground, the regular 18.2 HZ clock ticks arrive and are intercepted and/or disposed of in a DOS 3.3 compatible manner. When the DOS mode is in the background, the 18.2 HZ clock is masked off. The clock/calendar clock continues to run in both modes.

The CLOCK$ device itself -- the driver that sets and returns time of day -- is dual mode and services both modes. There is no reservation of the device; the time can be set from either mode.

The CLOCK device is unique because OS/2 reads or writes a 6-byte sequence which encodes the date and time. Writing to this device sets the date and time, and reading from it gets the date and time.

The following diagram illustrates the binary time format used by the CLOCK device:

CLOCK Device Time Format

Byte 0 Byte 1	Byte 2	Byte 3	Byte 4	Byte 5
Days since 1-1-80 Low-byte Hi-byte	Minutes	Hours	Sec/100	Seconds

The CLOCK$ device driver sets and maintains the following fields in the Global InfoSeg:

TIME

Time from 1-1-1970 in seconds
Milliseconds
Hours
Minutes
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 remain synchronized with the CMOS clock and that the hundredths of seconds field is correctly synchronized with the seconds field.

Console Device Drivers (Screen and Keyboard)

In OS/2, the generic console device driver has been replaced by two independent device drivers:

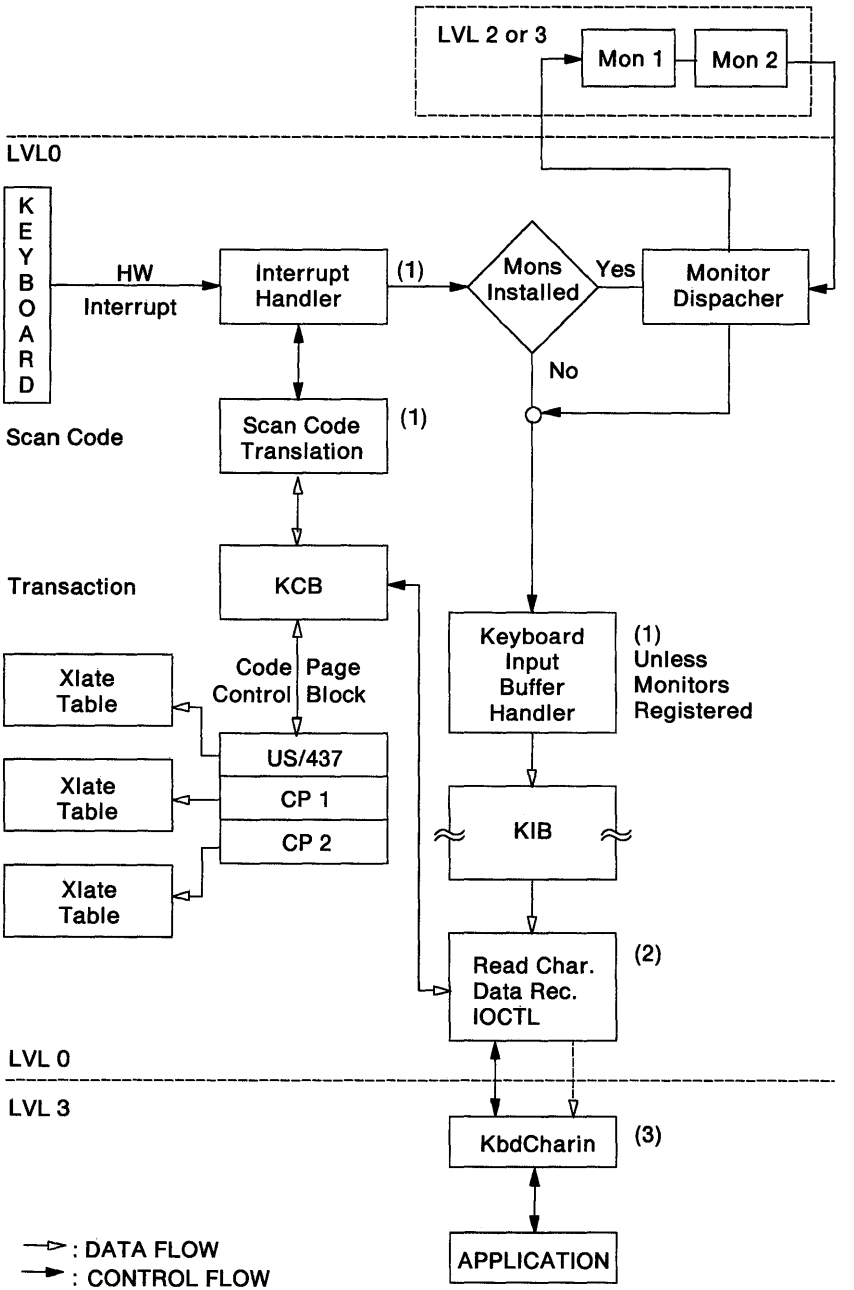
Screen or console output (SCR$)
Keyboard input (KBD$).

The KBD$ device driver supports the OS/2 interrupt-driven architecture. The SCR$ and KBD$ device drivers are part of the resident device driver set.

Keyboard Device Driver KBD$

The keyboard device driver interfaces the physical keyboard to applications through various levels of routines.

Keyboard System Structure



Note:

(1) Is part of the device driver interrupt handler
(2) Is part of the device driver strategy routine
(3) Is part of the base sub-system/router

The device driver receives the make-and-break keystroke scan codes and performs one or more of the following operations:

- Translates the scan code to an ASCII character
- Recognizes the key as a special key and signals the appropriate routine for processing
- Passes the key to the monitor dispatcher for further custom processing by monitors
- Places the key character data record into the appropriate session's keyboard input buffer.
- Calls the following sub-system functions to support handles:

  KbdOpen - Create a new logical keyboard

  KbdClose - Delete a logical keyboard

  KbdGetFocus - Bind the real keyboard to the logical one

  KbdFreeFocus - Free the real to logical keyboard bind.

The keyboard device driver supports code page switching. The following is a result of this support:

- Each handle may use one of two system-wide code pages. One is the current and the other may be swapped to. Each handle may also use the PC US 437 code page.
- The code pages to use are defined in CONFIG.SYS with the CODEPAGE and DEVINFO commands.
- The code pages are for one language only. Code pages for different languages could result if default code pages are not accepted during execution of the KEYB command. This could occur when attempting to load translate tables in a nonsupported code page for that language.
- The user is allowed, via the KEYB command, to change the language layout of the keyboard.

- The user may control, by the KbdSetCp sub-system function or the CHCP command, which of the two code pages is used in the handle for translation.
- Two sub-system functions support code page switching:

  KbdSetCp - Set to an installed code page, load if necessary.

  KbdGetCp - Get the current in-use code page.
- KbdSetCustXt - Adds a custom code page option
- KbdXlate - Translates a scan code
- KbdSetCp, KbdGetCp,and KbdXlate may be replaced in the sub-system by the use of the KbdRegister function.
- The code page header contains the code page number, the language ID of the table, the language default table indicator, and the keyboard type indicator.

Keyboard Initialization

Prior to CONFIG.SYS processing, the device driver and keyboard sub-system routines are loaded. As part of the device driver load, the keyboard initialization routine is executed. At this time level 0 memory is allocated and initialized. This memory will contain the KCBs, KIB, code pages, and related control blocks. A KCB is created for each possible session; this KCB is referred to as the 'default' logical keyboard for that session and is initialized to use the default code page.

At the end of CONFIG.SYS processing, after the CODEPAGE and DEVINFO statements have been processed, the code pages must be initialized. This is done by calling KbdSetCp.

Keyboard Run Time Operation

When a session is started, a default logical keyboard will already exist. This keyboard is identified to the sub-system by a 0 handle. Any program may share the keyboard by using handle 0.

If multiple programs use the default keyboard, they must coordinate their access to it. The default keyboard logically terminates when the session terminates.

If a program wants a logical keyboard separate from the default logical keyboard, it does a KbdOpen. This open creates a new logical keyboard, but does not make the physical to logical bond. As a result of an open, a handle unique within a process is returned to the caller which is later used to identify the logical keyboard. The handle ownership is tied to the process; handles are not inherited. Use of KbdOpen does not prohibit the process from using the default keyboard.

To make the physical to logical bond, the process issues the KbdGetFocus, using the handle identifying the logical keyboard. Once the bond is made, the logical keyboard may receive keystrokes. Note that type-ahead keystrokes are not possible before the bond is made. The Keyboard API may only be used when the bond is made or with handle 0 when no other handle has the bond.

The bond represents a foreground keyboard; one exists per session. This is either the default or a created logical keyboard.

Breaking the bond is done with the KbdFreeFocus call. If other threads have a KbdGetFocus outstanding, the thread having the highest priority will get the bond. If there are no KbdGetFocus calls outstanding, the physical keyboard will revert to the default keyboard.

A logical keyboard is destroyed with a KbdClose. The close will do a free focus, flush buffer, and de-allocate the KCB and related memory. Close will be done by the process kill mechanism, if not done by the program.

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, modify, or replace keystrokes. This is made possible by the keystroke monitor function.

When the DosMonReg call is used with keyboard devices, the **index** indicates the session, from 0 to 15 (see DosGetInfoSeg). -1 indicates the session of the calling thread.

The size of the keyboard device driver's data buffer is 16 bytes. This is the value to be used in calculating the sizes of the input/output buffers required for the DosMonReg call.

The keyboard device driver supports device monitors. The keyboard device driver passes its information to the monitors in packets which contain the following information:

Keystroke Monitor Data Packet Definition

MonFlagWord:		Word
CharDataRecord	XlatedChar:	Byte
	XlatedScan:	Byte
	DBCS Status:	Byte
	DBCS Shift	Byte
	Shift State:	Word
	Milliseconds:	DWord
KbdDDFlagWord		Word

MonFlagWord Lower Byte: Monitor Dispatcher Flags

Bit# ***Meaning***

7-3 RESERVED = 0
Should be passed untouched on packets being passed on. Should be set to 0 on packets that are being inserted by a monitor.

2 FLUSH
This is a flush packet. No other information in the packet has meaning. Monitor should flush its internal buffers and pass the packet quickly.

1 CLOSE
Not used by keystroke monitors.

0 OPEN
Not used by keystroke monitors.

MonFlagWord Upper Byte: Original Scan code, as read from the hardware. If 0, this packet was inserted for other reasons, see "KbdDDFlagWord" on page 9-95. Monitors pass this field untouched. Monitors should put a 0 here if they insert a packet.

CharData Record: Same as defined by the KbdCharIn function call in *Technical Reference, Vol. 2.*

KbdDDFlagWord

Bit# ***Meaning***

15,14 Available.
These bits are available for communication between monitors. They are not used by the device driver in any way. The monitor applications coordinate the use of these flags.

13-10 RESERVED = 0
Monitors should pass these flags as is. They should set these flags to 0 in packets that they create.

9 ACCENTED
This key was translated using the previous key passed, which was an accent key (Refer to 10H ACCENT KEY on page 9-98). In the case where an accent key is pressed and the following key doesn't use the accent, a packet containing the accent character itself is first passed, with this bit set (and the scan code field of MonFlagWord, see above, would be 0, indicating a non-key-generated record). Then a valid packet containing that following keystroke is passed, without this bit set.

8 MULTIMAKE
The translation process sees this scan code as a typeamatic repeat of a toggle key or a shift key. Because toggle and shift keys only change state on the first make after each key-break, no state information is changed (for example, the NUMLOCK toggle bit in the shift status word is not changed, even though this may be the NumLock key). If this key is a valid character, it will not go into the KIB once this bit is set.

7 SECONDARY
The scan code prior to the one in this packet was the SECONDARY KEY PREFIX (see below).

6 KEY BREAK
This record is generated by the release (the BREAK) of the key involved.

5-0 Numeric field that tells the device driver that this is a key that requires action. The number in this field is filled in during the translation of the scan-code. The value here allows the device driver to act on keystrokes without regard for what scan codes the keyboard uses or character codes that the current translation process may be using. The following values are currently defined:

VALUE FOR KEYS THAT ARE ALWAYS PLACED IN THE KIB

0 - No special action. Always place in KIB.

Values Acted On Prior to Passing Packet to Monitors

Except for the final keystroke of the REBOOT and DUMP key sequences, all of these values are passed on to the monitors. They will NOT be placed in the KIB. The XlatedChar and XlatedScan fields are undefined for these values:

Scan Code Meaning

01H ACK
This scan code was a keyboard acknowledge. Personal Computer AT attached keyboards would set this value on a FAH scan code.

02H SECONDARY KEY PREFIX
This scan code was a prefix scan code generated by the Enhanced Keyboard, indicating that the next scan code coming is one of the secondary keys that exists on that keyboard. Usually set on a E0H scan code or a hex E1 scan code.

03H KBD OVERRUN
This scan code was an over-run indication from the keyboard. On a Personal Computer AT attached keyboard, this value would be set on a FFH scan code.

04H RESEND
This scan code was a "resend" request from the keyboard. On a Personal Computer AT attached keyboard this value would be set on a FEH scan code.

05H REBOOT KEY
This scan code is the key that completes the multi-key restart sequence. On a Personal Computer AT attached keyboard this value would be used when the Ctrl + Alt + Delete sequence is seen.

06H DUMP KEY
This scan code is is the key that completes the multi-key Stand Alone Dump request sequence. On a Personal Computer AT attached keyboard, this value would be used on completion of the second consecutive press of Ctrl + Alt + NumLock without other keystrokes between the two presses.

07H-0AH See "Values Acted On After Passing Packet To Monitors."

0BH Invalid Accent Combination
This scan code is one that follows an accent scan code, but the combination was not valid and neither key is put in the KIB.

Note: This is set if the Canadian-French code pages are in use.

0C-0FH RESERVED = 0
(Will be treated as UNDEFINED, see entry 3FH.)

Values Acted On After Passing Packet To Monitors

Except where noted, these will be placed in the KIB when the device driver is in BINARY mode, but will not be placed in the KIB when the device driver is in ASCII mode. Also noted are those that never get placed in the KIB.

Scan Code Meaning

07H SHIFT KEY
This scan code translated as a shift key and has affected the shift status fields of the CharData record but does not generate a defined character, so it will not be placed in the KIB. The XlatedChar field is undefined.

08H PAUSE KEY
This scan code was translated as the key sequence meaning PAUSE. On a Personal Computer AT attached keyboard, this value will be used when the Ctrl + NumLock sequence is seen. The key itself will not be 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 ASCII mode. On most keyboards this would be when the Ctrl + S combination is seen. The key itself will not be placed in the KIB.

0AH WAKE-UP KEY

This scan code is the key that follows a PAUSE KEY or PSEUDO-PAUSE KEY which will cause the pause state to be ended. The key itself will be not be placed in the KIB.

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 hit, but it is not put into the KIB (unless the ACCENTED bit is on). Refer to ACCENTED bit on page 9-95. The next key determines this decision. If the next key is one that can be accented, then it will be passed by itself, with the ACCENTED bit on. If that next key cannot be accented by this accent, then two packets are passed. The first contains the character to print for the accent itself, has this value (ACCENT KEY) and has the ACCENTED flag (which says it's okay to put it in the KIB). The second packet contains a regular translation of that following key.

Note: The two packets get passed for every language except Canadian-French. See entry 0BH.

11H BREAK KEY

This scan code was translated as the key sequence meaning BREAK. On the Personal Computer AT attached keyboard, this value will be used where the Ctrl + Break sequence is seen.

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 ASCII mode. On most keyboards this would be when the Ctrl + C combination is seen. Note that the event generated by this key is separate from that generated by the BREAK KEY when in the binary mode.

13H PRINT SCREEN KEY

This scan code was translated as the key sequence meaning PRINT SCREEN. On a Personal Computer AT attached keyboard, this value will be used where the Shift-PrtSc sequence is seen.

14H PRINT ECHO KEY
This scan code was translated as the key sequence meaning PRINT ECHO. This value will be used where the Ctrl + PrtSc sequence is seen.

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 ASCII mode. On most keyboards this would show as the Ctrl + P combination.

16H PRINT-FLUSH KEY
This scan code is translated into the key sequence Print-Flush. This value will be used where the Ctrl + Alt + PrtSc sequence is seen.

17-2FH RESERVED = 0
Will be treated as undefined, see entry 3FH.

Values for Packets Not Generated by a Keystroke

Scan Code Meaning

30-37H RESERVED

38-3EH RESERVED
Will be treated as UNDEFINED, see entry 3FH.

Value for Keys that the Translation Process does not Recognize

Scan Code Meaning

3FH UNDEFINED
This scan code, or its combination with the current shift state, was not recognized in the translation process.

Special Key Processing

OS/2 examines each incoming keyboard character to determine if it should cause an asynchronous signal to be sent to some process (for example, Ctrl + C). The keyboard device driver responds to the following keystrokes:

Keys	Mode	Event Signalled or Signal Handler Called	Value Placed in KIB
Ctrl + C	Binary	Nothing	03H
Ctrl + C	ASCII	SIGINTR	Nothing
Ctrl + S	Binary	Nothing	13H
Ctrl + S	ASCII	event_CtrlScrLk	Nothing
Ctrl + P	Binary	Nothing	10H
Ctrl + P	ASCII	event_CtrlPrtSc	Nothing
Ctrl + Break	Binary	SIGBREAK	00:00H
Ctrl + Break	ASCII	SIGINTR	Nothing
Ctrl + NumLock	Binary	event_CtrlScrLk	Nothing
Ctrl + NumLock	ASCII	event_CtrlScrLk	Nothing
Ctrl + PrtSc	Binary	event_CtrlPrtSc	00:72H
Ctrl + PrtSc	ASCII	event_CtrlPrtSc	Nothing
PrtSc	Binary	event_ShftPrtSc	Nothing
PrtSc	ASCII	event_ShftPrtSc	Nothing

Notes:

1. The value xxH represents the translated ASCII field in the character data record.
2. The value xx:yyH represents the translated ASCII:translated scan code fields of the character data record.
3. Ctrl + NumLock has no effect on an IBM Enhanced keyboard; the Pause key provides this function.

Key	*Meaning*
Hot Key	Change sessions
Ctrl + Alt + Del	System IPL (Restart)
Ctrl + Alt + NumLock	Pressed twice means system dump
PrtScr	Print the current display screen (Shift + PrtScr on an IBM Personal Computer AT keyboard and Print Screen on an IBM Enhanced keyboard)

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 **Send_Event** helper function is invoked.

Note that the system hot key is defined by IOCtl Category 4 Function 56H (Set Session Manager Hot Key).

Compatibility Operations

The DOS mode runs in its own session; the session mechanism keeps OS/2 mode programs from affecting the compatibility screen. When the DOS mode is no longer in the foreground, the old application is frozen, keeping it from affecting the OS/2 screen. The DOS mode has, in effect, an independent SCRN and KBD device driver in the form of a DOS mode CON driver. This driver is compatible with the ROM INT10 and INT16 entries and their associated low-memory data structures.

EGA.SYS Device Driver

EGA.SYS is a device driver which provides support for the EGA Register Interface in the DOS mode. In the DOS mode, to support advanced graphics modes D, E, F and 10, the mouse pointer draw device driver must save/restore the EGA registers. Because the EGA registers are not readable, this can be done only if the application cooperates in setting the registers in the first place. Rather than doing I/O directly to the registers on the adapter, the application sets the registers through the EGA Register Interface.

EGA Register Interface

The EGA Register Interface is a library of ten functions supported for DOS mode, advanced graphics (modes D, E, F, and 10) applications. These functions do the following:

- Read from or write to one or more of the EGA write-only registers.
- Define default values for the EGA write-only registers, reset the EGA registers to these default values, or return the default values.
- Check whether the EGA Register Interface is present and, if so, return its version number.

When your application uses the EGA Register Interface, OS/2 maintains a backup copy of (shadows) how the EGA registers are set. Then, if the operator switches away from and later returns to your application, the registers will be restored properly.

It is not necessary to use the EGA Register Interface to set the mode, color palette, or palette registers. Instead, use ROM BIOS function call INT 10H with AH = 00H, 0BH, or 10H, respectively.

How to Call the EGA Register Interface: To call EGA Register Interface functions from an assembly language program:

1. Load the registers with the required parameter values.
2. Execute software interrupt 10h.

Values returned by the EGA Register Interface functions are placed in registers.

EGA Register Interface Restrictions: Functions not supported

Multiple display pages in graphics modes are not supported. Fonts may be loaded (using ROM BIOS INT 10H with AH = 11H) into character generator block 0 only.

Attribute Controller Registers

Before your application program uses the Attribute Controller registers (I/O address 3C0H) in an extended interrupt 10H call, it must set the flip-flop that selects the address or data register so that it selects the address register (by doing an input from I/O port 3BAH or 3DAH). The flip-flop is always reset to this state upon return from the extended interrupt 10H call.

Interrupt routines that access the attribute chip must also leave the flip-flop set to the address register upon return from the interrupt. Note, if your application program sets the flip-flop so that it selects the Data register and expects the flip-flop to remain in this state, your application must disable interrupts between the time it sets the flip-flop to the Data register state and the last time the flip-flop is assumed to be in this state.

Sequencer Memory Mode Register

When the Sequencer Memory Mode register (I/O address 3C5H, data register 4) is accessed, the sequencer produces a glitch on the CAS lines that may cause problems with video random access memory. As a result, your application program cannot use the EGA Register Interface to read from or write to this register. Instead, use the following procedure to safely alter this register:

1. Disable interrupts.
2. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 0.
3. Read/modify/write the Sequencer Memory Mode register.
4. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 1.
5. Enable interrupts.

Input Status Registers

Your application program cannot use the EGA Register Interface to read Input Status registers 0 (I/O address 3C2H) and 1 (I/O address 3BAH or 3DAH). If your program must read these registers, it should do so directly.

Graphics Controller Miscellaneous Register

When the Graphics Controller Miscellaneous register (I/O address 3CFH, data register 6) is accessed, a glitch on the CAS lines occurs that may cause problems with video random access memory. As a result, your application program should not use the EGA Register Interface to read from or write to this register.

EGA Register Interface function F6 does not alter the state of the Graphics Controller Miscellaneous register. Instead, use the following procedure to safely alter this register:

1. Disable interrupts.
2. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 0.
3. Read/modify/write the Graphics Controller Miscellaneous register.
4. Set Synchronous Reset (bit 1) in the Sequencer Reset register to 1.
5. Enable interrupts.

EGA Register Interface Functions: This section describes each EGA Register Interface function in detail. The following list shows these functions by function number:

Number (Hex)	***Function***
F0	Read one register
F1	Write one register
F2	Read register range
F3	Write register range
F4	Read register set
F5	Write register set
F6	Revert to default registers
F7	Define default register table
F8	Read default register table
FA	Interrogate driver

Note: Calls F9H, and FBH through FFH are reserved.

Each function description includes:

- The parameters required to make the call (input) and the expected return values (output).
- Any special considerations regarding the function.

If the function description does not specify an input for a parameter, you don't need to supply a value for that parameter before making the call. If the function description does not specify an output value for a parameter, the parameter's value is the same before and after the call.

Note: The EGA Register Interface does not check input values, so be sure that the values you load into the registers before making a call are correct.

Function F0 - Read One Register

Purpose Function F0 reads data from a specified register on the EGA.

Input:

AH = F0H

BX = Pointer for pointer/data chips:

BH = 0
BL = Pointer

Ignored for single registers

DX = Port number:
Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Output:

AX: Restored
BH: Restored
BL: Data
DX: Restored

All other registers restored.

Example: The following example saves the contents of the Sequencer Map Mask register in "myvalue:"

```
myvalue db   ?
        mov  ah, 0f0h        ; f0 = read one register
        mov  bx, 0002h       ; bh = 0 / bl = map mask
                             ; index
        mov  dx, 0008h       ; dx = sequencer
        int  10h             ; get it!
        mov  myvalue, bl     ; save it!
```

The following example saves the contents of the Miscellaneous Output register in "myvalue":

```
myvalue db   ?
        mov  ah, 0f0h        ; f0 = read one register
        mov  dx, 0020h       ; dx = miscellaneous
                             ;  output register
        int  10h             ; get it!
        mov  myvalue, bl     ; save it!
```

Function F1 - Write One Register

Purpose Function F1 writes data to a specified register on the EGA.

When your application program returns from a call to function F1, the contents of registers BH and DX are not restored. Your program must save and restore these registers if desired.

Input:

AH = F1h

BL = Pointer for pointer/data chips
Data for single registers,

BH = Data for pointer/data chips
(ignored for single registers)

DX = Port number:
Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Output:

AX: Restored
BL: Restored
BH: Not restored
DX: Not restored

All other registers restored.

Example: The following example writes the contents of "myvalue" into the CRT Controller Cursor Start register:

```
myvalue db   3h
        mov  ah, 0f1h        ; f1 = write one register
        mov  bh, myvalue     ; bh = data from myvalue
        mov  bl, 000ah       ; bl = cursor start index
        mov  dx, 0000h       ; dx = crt controller
        int  10h             ; write it!
```

The following example writes the contents of "myvalue" into the Feature Control register:

```
myvalue db   2h
        mov  ah, 0f1h        ; f1 = write one register
        mov  bl, myvalue     ; bl = data from myvalue
        mov  dx, 0028h       ; dx = feature control register
        int  10h             ; write it!
```

Function F2 - Read Register Range

Purpose Function F2 reads data from a specified range of registers on the EGA. A range of registers is defined to be several registers on a single chip that have consecutive indexes. This call is applicable for pointer/data chips.

Input:

AH = F2h

CH = Starting pointer value

CL = Number of registers (must be > 1)

DX = Port number:

0h: CRT Controller (3?4h)
8h: Sequencer (3C4h)
10h: Graphics Controller (3CEh)
18h: Attribute Controller (3C0h)

? = B for monochrome modes or D for color modes

ES:BX = Points to table of one-byte entries (length = value in CL). On return, each entry is set to the contents of the corresponding register.

Output:

AX: Restored
BX: Restored
CX: Not restored
DX: Restored
ES: Restored

All other registers restored.

Example: The following example saves the contents of the Attribute Controller Palette registers in "paltable:"

```
paltable db  16 dup (?)
         mov  ax, ds               ; assume paltable in
                                   ;  data segment
         mov  es, ax               ; es = data segment
         mov  bx, offset paltable  ; es:bx = paltable
                                   ;  address
         mov  ah, 0f2h             ; f2 = read register
                                   ;  range
         mov  cx, 0010h            ; ch = start index
                                   ;  of 0
                                   ; cl = 16 registers
                                   ;  to read
         mov  dx, 0018h            ; dx = attribute
                                   ;  controller
         int  10h                  ; read them!
```

Function F3 - Write Register Range

Purpose Function F3 writes data to a specified range of registers on the EGA. A range of registers is defined to be several registers on a single chip that have consecutive indexes. This call is applicable for the pointer/data chips.

Input:

AH = F3h

CH = Starting pointer value

CL = Number of registers (must be > 1)

DX = Port number:

0h: CRT Controller (3?4h)
8h: Sequencer (3C4h)
10h: Graphics Controller (3CEh)
18h: Attribute Controller (3C0h)

? = B for monochrome modes or D for color modes

ES:BX = Points to table of one-byte entries (length = value in CL). Each entry contains the value to be written to the corresponding register.

Output:

AX: Restored
BX: Not restored
CX: Not restored
DX: Not restored
ES: Restored

All other registers restored

Example: The following example writes the contents of "cursloc" into the CRT Controller Cursor Location High and Cursor Location Low registers.

```
cursloc db   01h, 00h            ; cursor at page
                                 ;  offset 0100h
        mov  ax, ds              ; assume cursloc in
                                 ;  data segment
        mov  es, ax              ; es = data segment
        mov  bx, offset cursloc  ; es:bx = cursloc
                                 ;  address
        mov  ah, 0f3h            ; f3 = write register
                                 ;  range
        mov  cx, 0e02h           ; ch = start index
                                 ;  of 14
                                 ; cl = 2 registers to
                                 ;  write
        mov  dx, 0000h           ; dx = crt controller
        int  10h                 ; write them!
```

Function F4 - Read Register Set

Purpose Function F4 reads data from a set of registers on the EGA. A set of registers is defined to be several registers that may or may not have consecutive indexes, and that may or may not be on the same chip.

Input:

AH = F4H

CX = Number of registers (must be > 1)

ES:BX = Points to table of records with each entry in this format:

Bytes 1-2: Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Byte 3: Pointer value (0 for single registers)

Byte 4: EGA Register Interface fills in data read from register specified in bytes 1-3.

Output:

AX: Restored
BX: Restored
CX: Not restored
ES: Restored

All other registers restored

Example: The following example saves the contents of the Miscellaneous Output register, Sequencer Memory Mode register, and CRT Controller Mode Control register in "results:"

```
outvals dw   0020h        ; miscellaneous output
                          ;  register
        db   0            ; 0 for single registers
        db   ?            ; returned value
        dw   0008h        ; sequencer
        db   04h          ; memory mode register
                          ;  index
        db   ?            ; returned value
        dw   0000h        ; crt controller
        db   17h          ; mode control register
                          ;  index
        db   ?            ; returned value

results db   3 dup (?)
        mov  ax, ds       ; assume outvals in
                          ;  data segment
        mov  es, ax       ; es = data segment
        mov  bx, offset outvals  ; es:bx = outvals
                                 ;  address
        mov  ah, 0f4h     ; f4 = read register set
        mov  cx, 3        ; number of entries in
                          ;  outvals
        int  10h          ; get values into outvals
        mov  si, 3              ; move the returned
                                ;  values from
        add  si, offset outvals  ;  outvals
        mov  di, offset results  ;  to results
        mov  cx, 3        ; 3 values to move

loop:   mov  al, [si]     ; move one value from
        mov  [di], al     ;  outvals to results
        add  si, 4        ; skip to next source byte
        inc  di           ; point to next destination
                          ;  byte
        loop loop
```

Function F5 - Write Register Set

Purpose Function F5 writes data to a set of registers on the EGA. A set of registers is defined to be several registers that may or may not have consecutive indexes, and that may or may not be on the same chip.

Input:

AH = F5h

CX = Number of registers (must be > 1)

ES:BX = Points to table of values with each entry in this format:

Bytes 1-2: Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

Byte 3: Pointer value (0 for single registers)

Byte 4: Data to be written to register specified in bytes 1-3

Output:

AX: Restored
BX: Restored
CX: Not restored
ES: Restored

All other registers restored

Example: The following example writes the contents of "outvals" to the Miscellaneous Output register, Sequencer Memory Mode register, and CRT Controller Mode Control register:

```
outvals dw   0020h        ; miscellaneous output
                          ;  register
        db   0            ; 0 for single registers
        db   0a7h         ; output value
        dw   0008h        ; sequencer
        db   04h          ; memory mode register index
        db   03h          ; output value
        dw   0000h        ; crt controller
        db   17h          ; mode control register index
        db   0a3h         ; output value
        mov  ax, ds       ; assume outvals in
                          ;  data segment
        mov  es, ax       ; es = data segment
        mov  bx, offset outvals  ; es:bx = outvals
                                 ;  address
        mov  ah, 0f5h     ; f5 = write register set
        mov  cx, 3        ; number of entries in
                          ;  outvals
        int  10h          ; write the registers!
```

Function F6 - Revert to Default Registers

Purpose Function F6 restores the default settings of any registers that your application program has changed through the EGA Register Interface. The default settings are defined in a call to function F7 (described in the next section).

Input:

AH = F6h

Output:

All registers restored

Example: The following example restores the default settings of the EGA registers:

```
        mov  ah, 0f6h     ; f6 = revert to default
                          ;  registers
        int  10h          ; do it now!
```

Function F7 - Define Default Register Table

Purpose Function F7 defines a table containing default values for any pointer/data chip or single register. If you define default values for a pointer/data chip, you must define them for all registers within that chip.

Input:

AH = F7h

DX = Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

ES:BX = Points to table of one-byte entries. Each entry contains the default value for the corresponding register. The table must contain entries for all registers.

Output:

AX: Restored
BX: Not restored
DX: Not restored
ES: Restored

All other registers restored

Example: The following example defines default values for the Attribute Controller:

```
attrdflt db  00h, 01h, 02h, 03h, 04h, 05h, 06h, 07h
         db  10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h
         db  08h, 00h, 0fh, 00h
         mov  ax, ds              ; assume attrdflt in
                                  ;  data segment
         mov  es, ax              ; es = data segment
         mov  bx, offset attrdflt ; es:bx = attrdflt
                                  ;  address
         mov  ah, 0f7h            ; f7 = define default
                                  ;  register table
         mov  dx, 0018h           ; dx = attribute
                                  ;  controller
         int  10h                 ; do it!
```

The following example defines a default value for the Feature Control register:

```
featdflt db  00h
         mov  ax, ds              ; assume featdflt in
                                  ;  data segment
         mov  es, ax              ; es = data segment
         mov  bx, offset featdflt ; es:bx = featdflt
                                  ;  address
         mov  ah, 0f7h            ; f7 = define default
                                  ;  register table
         mov  dx, 0028h           ; dx = feature
                                  ;  control register
         int  10h                 ; do it!
```

Function F8 - Read Default Register Table

Purpose Function F8 reads the table containing default register values for any pointer/data chip or single register.

Input:

AH = 0F8H

DX = Port number:

Pointer/data chips

0h: CRT Controller (3?4H)
8h: Sequencer (3C4H)
10h: Graphics Controller (3CEH)
18h: Attribute Controller (3C0H)

Single registers

20h: Miscellaneous Output register (3C2H)
28h: Feature Control register (3?AH)
30h: Graphics 1 Position register (3CCH)
38h: Graphics 2 Position register (3CAH)

? = B for monochrome modes or D for color modes

ES:BX = Points to a table into which the default values are returned. The table must have room for the full set of values for the pointer/data chip or single register specified.

Output:

AX: Restored
BX: Not restored
DX: Not restored
ES: Restored

All other registers restored

Function FA - Interrogate Driver

Purpose Function FA returns a value specifying whether the EGA.SYS driver is present.

Input:

AH = FAh

BX = 0

Output:

AX: Restored
BX: 0, if EGA.SYS driver is not present
ES:BX: Pointer to EGA Register Interface version number, if present:
Byte 1: Major release number
Byte 2: Minor release number (in 1/100ths)

Example: The following example interrogates the driver and displays the results:

```
gotmsg  db   "EGA.SYS driver found", 0dh, 0ah, 24h
nopmsg  db   "EGA.SYS driver not found", 0dh, 0ah, 24h
revmsg  db   "revision $"
crlf    db   0dh, 0ah, 24h
ten     db   10
        mov  bx, 0          ; must be 0 for this call
        mov  ah, 0fah       ; fa = interrogate driver
        int  10h            ; interrogate!
        or   bx, bx         ; bx = 0 ?
        jnz  found          ; branch if driver present
        mov  dx, offset nopmsg  ; assume nopmsg in data
                                ;  segment
        mov  ah, 09h        ; 9 = print string
        int  21h            ; output not found message
        jmp  continue       ; that's all for now
found:  mov  dx, offset gotmsg  ; assume gotmsg in data
                                ;  segment
        mov  ah, 09h            ; 9 = print string
        int  21h                ; output found message
        mov  dx, offset revmsg  ; assume revmsg in data
                                ;  segment
        mov  ah, 09h        ; 9 = print string
        int  21h            ; output "revision "
        mov  dl, es:[bx]    ; dl = major release
                            ;  number
```

```
        add  dl, "0"          ; convert to ASCII
        mov  ah, 2            ; 2 = display character
        int  21h              ; output major release
                              ;  number
        mov  dl, "."          ; dl = "."
        mov  ah, 2            ; 2 = display character
        int  21h              ; output a period
        mov  al, es:[bx+1]    ; al = minor release
                              ;  number
        xor  ah, ah           ; ah = 0
        idiv ten              ; al = 10ths, ah = 100ths
        mov  bx, ax           ; save ax in bx
        mov  dl, al           ; dl = 10ths
        add  dl, "0"          ; convert to ASCII
        mov  ah, 2            ; 2 = display character
        int  21h              ; output minor release
                              ;  10ths
        mov  dl, bh           ; dl = 100ths
        add  dl, "0"          ; convert to ASCII
        mov  ah, 2            ; 2 = display character
        int  21h              ; output minor release
                              ;  100ths
        mov  dx, offset crlf; assume crlf in data
                              ;  segment
        mov  ah, 09h          ; 9 = print string
        int  21h              ; output end of line
continue:                     ; the end
```

Using Extended Screen and Keyboard Control (ANSI.SYS, ANSICALL.DLL)

This section explains how you can issue special control character sequences to:

- Control the position of the cursor
- Erase text from the screen
- Set the display mode
- Redefine the meaning of keyboard keys

ANSI extended screen and keyboard control sequences are supported in the DOS mode by ANSI.SYS, an installable device driver.

In the OS/2 mode, these control sequences are supported by ANSICALL.DLL, a dynamic link module.

Note: In this section, unless otherwise specified, ANSI refers to both ANSI.SYS and ANSICALL.DLL.

Limitations/Restrictions

ANSI operates on a per-session basis.

OS/2 mode ANSI is affected when keys are reassigned in a code page environment. ANSI does not provide code page support for key reassignment in the DOS mode.

Control Sequences

Control Sequence Syntax

Each of the cursor control sequences is in the format:

ESC [*parameters* COMMAND

ESC	The 1-byte ASCII code for ESC (1BH). It is not the three characters ESC.
[	The character [.
parameters	The numeric values you specify for #. The # represents a numeric parameter. A numeric parameter is an integer value specified with ASCII characters. If you do not specify a parameter value, or if you specify a value of 0, the default value for the parameter is used.
COMMAND	An alphabetic string that represents the command. It is case specific.

For example:

```
ESC [2;10H
```

could be created using BASIC as follows:

```
The IBM Personal Computer Basic
Version 3.00 Copyright IBM Corp. 1981, 1982, 1983, 1984
xxxxx Bytes free

Ok
open "sample" for output as 1
Ok
print #1, CHR$(27);"[2;10H";"x row 2 col 10"
Ok
close #1
Ok
```

Notice that "CHR$(27)" is ESC.

Cursor Control Sequences

The following tables contain the cursor control sequences you can use to control cursor positioning.

Cursor Position

Cursor Position	Function
ESC [#;#H	Moves the cursor to the position specified by the parameters. The first parameter specifies the row number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position.

This example copies the file SAMPLE from the previous example, to CON, which places the cursor on row 2 column 10 of the screen:

```
type sample
```

Cursor Up

Cursor Up	Function
ESC [#A	Moves the cursor up one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. This sequence is ignored if the cursor is already on the top line.

Cursor Down

Cursor Down	Function
ESC [#B	Moves the cursor down one or more rows without changing the column position. The value of # determines the number of lines moved. The default value for # is 1. The sequence is ignored if the cursor is already on the bottom line.

Cursor Forward

Cursor Forward	Function
ESC [#C	Moves the cursor forward one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored if the cursor is already in the rightmost column.

Cursor Backward

Cursor Backward	Function
ESC [#D	Moves the cursor back one or more columns without changing the row position. The value of # determines the number of columns moved. The default value for # is 1. This sequence is ignored if the cursor is already in the leftmost column.

Horizontal and Vertical Position

Horizontal and Vertical Position	Function
ESC [#;#f	Moves the cursor to the position specified by the parameters. The first parameter specifies the line number and the second parameter specifies the column number. The default value is 1. If no parameter is given, the cursor is moved to the home position.

Cursor Position Report

Cursor Position Report	Function
ESC [#;#R	The cursor sequence report reports the current cursor position through the standard input device. The first parameter specifies the current line and the second parameter specifies the current column.

Device Status Report

Device Status Report	Function
ESC [6n	The console driver gives a cursor position report sequence on receipt of device status report.

Note: Do not use the Device Status Report as part of a prompt.

This example tells ANSI to put the current cursor position (row and column) in STDIN. Then the program reads it from STDIN and outputs to STDOUT.

```
PROGRAM dsr(INPUT,OUTPUT);

  VAR
    f:FILE OF CHAR;
    key:CHAR;

  FUNCTION inkey:CHAR;                  { read character  }
    VAR                                 {  from the       }
      ch:CHAR;                          { keyboard buffer }
    BEGIN
      READ(f,ch);
      inkey:=ch
    END;

  BEGIN
    ASSIGN(f,'user');
    RESET(f);
        WRITE(CHR(27),'[6n');           { issue a DSR   }
        key:=inkey;                     { read up to    }
        key:=inkey;                     { first digit   }
        key:=inkey;                     { of the row    }
        WRITE('row ',inkey,inkey,'  column ');
        key:=inkey;                     { skip to column}
        WRITE(inkey,inkey)              { write column  }
  END.
```

Save Cursor Position

Save Cursor Position	Function
ESC [s	The current cursor position is saved. This cursor position can be restored with the restore cursor position sequence (see below).

Restore Cursor Position

Restore Cursor Position	Function
ESC [u	Restores the cursor to the value it had when the console driver received the save cursor position sequence.

Erasing

The following tables contain the control sequences you can use to erase text from the screen.

Erase in Display

Erase in Display	Function
ESC [2J	Erases all of the screen and the cursor goes to the home position.

Erase in Line

Erase in Line	Function
ESC [K	Erases from the cursor to the end of the line and includes the cursor position.

Controlling Display Mode

The following tables contain the control sequences you can use to set the mode of operation. They are:

- Set Graphics Rendition (SGR)
- Set Mode (SM)
- Reset Mode (RM)

Set Graphics Rendition (SGR)

SGR	Function
ESC [#;...;#m	Sets the character attribute specified by the parameters. All following characters have the attribute according to the parameters until the next occurrence of SGR. **Parameter Meaning** 0 All attributes off (normal white on black) 1 Bold on (high intensity) 4 Underscore on (mono-compatible modes) 5 Blink on 7 Reverse video on 8 Canceled on (invisible) 30 Black foreground 31 Red foreground 32 Green foreground 33 Yellow foreground 34 Blue foreground 35 Magenta foreground 36 Cyan foreground 37 White foreground 38 Reserved 39 Reserved 40 Black background 41 Red background 42 Green background 43 Yellow background 44 Blue background 45 Magenta background 46 Cyan background 47 White background

Set Mode (SM)

SM	Function
ESC [=#h or ESC [=h or ESC [=0h or ESC [?7h	Invokes the screen width or type specified by the parameter. **Parameter Meaning** 0 40x25 black and white 1 40x25 color 2 80x25 black and white 3 80x25 color 4 320x200 color 5 320x200 black and white 6 640x200 black and white 7 Wrap at end of line. (Typing past end-of-line results in new line.)

Reset Mode (RM)

RM	Function
ESC [=#l or ESC [=l or ESC [=0l or ESC [?7l	Parameters are the same as Set Mode (SM) except that parameter 7 resets wrap at end-of-line mode (characters past end-of-line are thrown away).

Keyboard Key Reassignment

When the application does a KbdStringIn call, the reassigned key's ASCII code is converted to the specified string and is passed back to the calling application.

OS/2 mode ANSI is affected when keys are reassigned in a code page environment. ANSI "remembers" the code page under which a key is reassigned. The keyboard subsystem checks for reassigned keys when the application calls the KbdStringIn function. When a reassigned key is detected, the ANSI support:

- Checks to see what code page the requestor is running under.
- Looks internally to see if the key has been reassigned under that code page.
- If there is a key reassignment for that code page, gives the reassignment string.
- Otherwise, gives the original ASCII codes.

A maximum storage of 64Kb may be allocated to OS/2 mode ANSI reassigned key definitions.

The following table contains the control sequences you can use to redefine the meaning of keyboard keys.

The control sequence is:	Function
ESC [#;#;...#p or ESC ["string"p or ESC [#;"string";#; #;"string";#p or any other combination of strings and decimal numbers	The first ASCII code in the control sequence defines which code is being mapped. The remaining numbers define the sequence of ASCII codes generated when this key is intercepted. However, if the first code in the sequence is 0 (NULL) the first and second code make up an extended ASCII redefinition

Here are some examples:

To execute these examples, you can either:

- Create a file that contains the following statements and then use the TYPE command to display the file that contains the statement.
- Execute the command at the OS/2 prompt.

1. Reassign the **Q** and **q** key to the **A** and **a** (and the other way as well):

   Creating a File:

   ```
   ESC [65;81p     A becomes Q
   ESC [97;113p    a becomes q
   ESC [81;65p     Q becomes A
   ESC [113;97p    q becomes a
   ```

   At the OS/2 Prompt:

   ```
   prompt $e[65;81p     A becomes Q
   prompt $e[97;113p    a becomes q
   prompt $e[81;65p     Q becomes A
   prompt $e[113;97p    q becomes a
   ```

2. Reassign the F10 key to a DIR command followed by a carriage return:

   Creating a File:

   ```
   ESC [0;68;"dir";13p
   ```

   At the OS/2 Prompt:

   ```
   prompt $e[0;68;"dir";13p
   ```

   The $e is the prompt command characters for ESC. The 0;68 is the extended ASCII code for the F10 key; 13 decimal is a carriage return.

3. The following example sets the prompt to display the current directory on the top of the screen and the current drive on the current line.

   ```
   prompt $e[s$e[1;30f$e[K$p$e[u$n$g
   ```

   If the current directory is C:\FILES, and the current drive is C, this example would display:

   ```
                   C:\FILES

   C>
   ```

4. The following DOS mode assembly language program reassigns the F10 key to a DIR B: command followed by a carriage return.

```
        TITLE SETANSI.ASM - SET F10 TO STRING FOR ANSI.SYS
CSEG    SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:CSEG,DS:CSEG
        ORG     100H
ENTPT:  JMP     SHORT START
STRING  DB      27,'[0;68;"DIR B:";13P' ;Redefine F10 key
STRSIZ  EQU     $-STRING                ;Length of above message
HANDLE  EQU     1                       ;Pre-defined file
                                        ;Handle for standard output

START   PROC    NEAR
        MOV     BX,HANDLE               ;Standard output device
        MOV     CX,STRSIZ               ;Get size of text to be sent
        MOV     DX,OFFSET STRING        ;Pass offset of string
                                        ;To be sent
        MOV     AH,40H                  ;Function="write to device"
        INT     21H                     ;Call DOS
        RET                             ;Return to DOS
START   ENDP

CSEG    ENDS
        END     ENTPT
```

Diskette Device Driver

The floppy disk device driver is able to run in a multitasking, dual-mode environment.

The following functions are provided.

- Read
- Write

  Reading and writing can be done 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.

- Verify
- Format

  With this function the user is able to determine if the drive is present, if the direct access storage device (DASD) is a diskette with change line available/not available, or if the DASD is a fixed disk.

- Get/Set Device Parameters

Fixed Disk Device Driver

The fixed disk device driver is able to run in a multitasking, dual mode environment.

The following functions are provided.

- Read
- Write

  Read and write requests can 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 cylinder and head and sector number to be specified.

- Verify
- Format
- Get/Set Drive Parameters

  With this function the user is able to determine the number of consecutive drives that are attached, the maximum usable value for head number, the maximum usable value for cylinder number, and the maximum usable value for sector number.

  The user is able to determine if a DASD is present, if the DASD is a diskette with change line available/not available, or if the DASD is a fixed disk. If the DASD is a fixed disk, the number of 512-byte blocks contained is returned. In addition, the disk device driver has some special support code for DOS mode applications which can issue INT 13H, which also runs in user state.

Greater than 32Mb Partitioned support

Large fixed disk (greater than 32Mb) are supported by OS/2 with partitioning of the disk. The extended partition is indicated by a system ID byte of 05H in the partition table of the Master Boot (Start-up) Record. This partition cannot be started, and programs that can set startable partitions (such as OS/2 FDISK) will not allow the partition to be marked as able to start.

Note: This extended partition support can be used on any fixed disk supported by OS/2.

The extended DOS partition can be created only if a primary DOS partition already exists on a startable drive. A Primary DOS partition is a partition with a system ID byte of 01H or 04H. If the drive cannot be started, then an extended DOS partition may be created without having a primary DOS partition.

The Extended DOS Partition starts and ends on a cylinder boundary.

Extended DOS Partition Architecture

The Extended DOS Partition consists of a collection of extended volumes which are linked together by a pointer in the extended volumes' extended start-up record. An extended volume consists of an extended start-up record and one logical block device. An extended volume created within the extended DOS partition can be any size from one cylinder long up through the maximum available contiguous space in the extended DOS partition. However, in OS/2 an extended volume cannot be larger than 32Mb due to the limitations of the FAT file system. All extended volumes must start and end on a cylinder boundary. An extended volume will correspond to an image of a physical disk. The extended start-up record corresponds to the master start-up record at the beginning of an actual physical disk and the logical block device corresponds to the DOS partition that is pointed to by the master start-up record.

Therefore, the logical block device begins with a normal DOS start-up sector if it is a DOS logical block device (system id= 1 or 4). This logical block device must start on a cylinder and head boundary and follows the extended start-up record on the physical disk. The logical block device and the extended volume both end on the same cylinder boundary.

Each extended volume contains an extended start-up record, located in the first sector of the disk location assigned to it. This extended start-up record contains the 55AAH signature ID byte. This allows programs that look at the extended (master) start-up record to be compatible. This extended start-up record also contains a partition table, which can contain only 2 types of entries. The start-up code is not critical, as the devices are not considered startable. The start-up code may simply report a message indicating an unstartable partition if it is executed.

The partition table portion of the extended start-up record is the same as the partition table structure in the master start-up record. This structure has four partition entries of 16 bytes each. The system ID byte must be filled in for all four entries with one of the following values:

00H No space allocated in this entry.

01H DOS partition up to 16Mb

04H DOS partition with 32Mb > SIZE > 16Mb

05H Maps out area assigned to the next extended volume. Serves as a pointer to the next extended start-up record.

06H Reserved.

If the system ID byte is 0 then the values in that partition table entry will be set to 0.

If OS/2 detects any values other than 01H or 04H, it will ignore that entry and not attempt to install the logical block device. This will allow future expansion of devices in this area without problems of compatibility with earlier systems.

The partition start and end fields (C,H,S) will be filled in for any of the four partition entries in an extended start-up record that have one of the above system ID bytes. This will allow a program such as FDISK to determine the allocated space in the extended DOS partition, as well as allowing the device drivers to determine the physical DASD area that belongs to it. The partition start and end fields (C,H,S) for the partition entry that points to the logical block device (system ID 01H, 04H, or 06H) map out the physical boundaries of the logical block device. They are offset relative to the beginning of the extended start-up record that the entry resides in. The partition start and end fields (C,H,S) for the partition entry that points to the next extended volume (system ID 05H) map out the physical boundaries of the next extended volume. They are relative to the beginning of the entire physical disk.

The relative sector and number of sector fields will be set up differently depending on what system ID byte is used. If 01H, 04H or 06H is in the system ID field for that extended partition entry (pointer to the logical block device), the relative sector field will be set up as an offset from (and including) the start of the extended start-up record for the associated extended volume. The number of sectors (size) field

will be filled in with the size of the created logical block device area, in other words, the number of sectors mapped out by the start and stop cylinder/track/sector fields. The size of the extended volume can be calculated by adding the relative sector field and the sector size field of the associated extended start-up record.

If the system ID byte is 05H, then the relative sector field is the offset (of the next extended volume) in sectors from the start of the entire extended DOS partition. The number of sectors field is not used in this field, and will be filled with 00H's.

This architecture allows only one logical block device to be defined for each extended start-up record. Therefore, only a maximum of two partition entries at a time is used in each extended start-up record; an entry with system ID byte of (01H, 04H, or 06H) and an entry with ID of 05H, which is the pointer to the next extended volume. Although only two entries can be used, a program installing these devices will not assume that the first two entries will be the non-zero entries.

Installing Block Devices in the Extended Partition

To install block devices, the device drivers will first install the primary DOS partitions on both the 80H and 81H physical drives if any exist. This will insure that an existing drive letter (D:) on the 81H drive will remain the same. After these devices are installed, on the 80H drive, the drivers will look for the existence of the extended DOS partition. If one exists, then it will look at the first sector of the extended DOS partition for the first extended start-up record. If there is a valid system ID (01h, 04h, or 06h) in any of the four partition entries, the device is installed and assigned the next available drive letter. This will occur before any CONFIG.SYS device drivers are loaded so the FDISK will correctly display the drive letter when space is allocated for the drive.

The first extended start-up record (in the extended DOS partition) is a special case because it is possible there will not be a device to be installed defined in the partition table. The first device might have been created and then deleted at some time, but the first extended start-up record is needed to point to the next one, if one exists. Any other extended start-up record will always have a device to be installed.

An extended start-up record may not contain a device that will be installed by that driver. For example, a logical block device with a system ID byte of 06H would not be installed in OS/2.

Once a device has been installed (or the special cases above occurs), the device driver will search the other partition entries for a system ID byte of 05H, indicating that another device (extended volume) exists. If a 05H is not found, there are no more logical block devices (extended volumes) in the extended DOS partition.

If a 05H system ID is found, the start location in that partition entry will be read in order to find the location of the next extended start-up record (extended volume). When located, it will be read in and then the process repeated in order to install additional devices.

Once all the valid devices for a physical drive have been installed, the next physical drive will be examined and the entire process repeated.

A device driver will not assume any order dependency when searching for a particular system ID byte in an extended start-up record. All four possible entries in a extended start-up record partition table will be searched before a driver decides that a particular system ID byte does not exist.

Creating Block Devices in the Extended DOS Partition

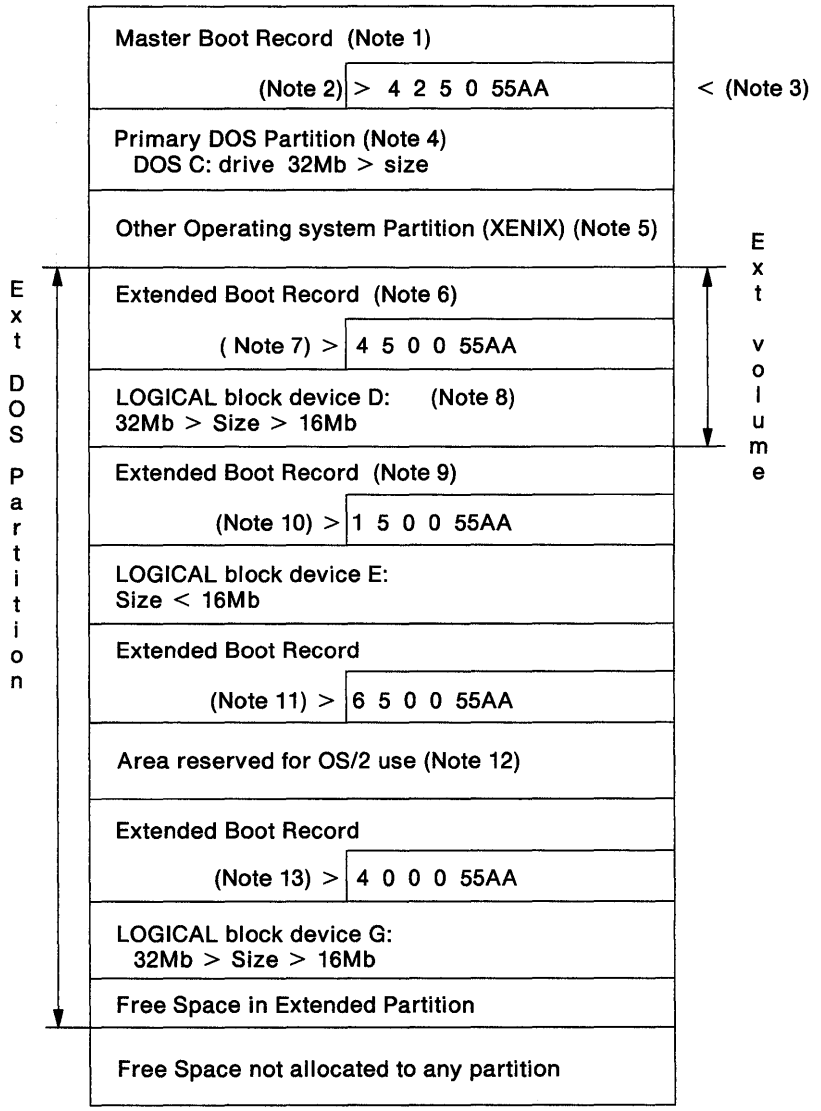
To create the structure for an extended volume in the extended DOS partition, FDISK will determine if there is available space in the extended DOS partition and if less than 24 total devices are allocated in the system. The maximum number of block devices allowed is 26, and two are used by diskettes A: and B:. If so, then the program will create an extended start-up record at the space located, with a partition entry filled in with the size and location information for that logical block device. If this is not the first extended start-up record, the program will back up to the last extended start-up record in the chain (as linked by the 05H entries), and create a partition entry in that extended start-up record that has the size and location data for the newly created record. This action will create the pointer required to locate the newly created start-up record.

If this is the first extended start-up record (in the extended DOS partition), only the size, type and location of the logical block device need to be put into a partition entry. The start of the extended DOS partition in the master start-up record will serve as a pointer to this extended volume.

Deleting Block Devices in the Extended DOS Partition

To delete a block device, the program will set to 0 the 16-byte partition entry that contained the system ID byte that indicated the device type (01H, 04H, or 06H). Also, if in the same extended start-up record there exists a partition entry with system ID of 05H, indicating that another extended volume exists, this information will be copied to the 05H partition entry of the previous extended start-up record. There is one exception to this rule: if the logical block device deleted is at the beginning of the extended DOS partition, only the partition entry indicating the device type would be set to 0. The 05h pointer information will be left in place.

Layout of Block Devices in the Extended DOS Partition



Note 1 Master start-up (boot) record code, starting at Trk 000, Hd 00, Sec 01 of disk 80H or 81H.

Note 2 Partition table for master start-up record. See "Partition Table for Master Start-up Record" on page 9-146 for layout. The 4 is the system ID byte in the partition table that indicates a DOS partition bigger than 16Mb, the 2 is a XENIX partition, and the 05H maps the extended DOS partition.

Note 3 55AAH is the signature to validate the master start-up record.

Note 4 Primary DOS area, must reside entirely in first 32Mb of disk. C: is block device 80H. D: is block device 81H if it exists. This partition has a maximum size of 32Mb.

Note 5 Other operating system on disk; XENIX in this example.

Note 6 Extended start-up record for extended volume that corresponds to logical block device D:. (This assumes only 80H block device exists.) If 81H block device exists, then this would be block device E:.

Note 7 Logical block device D: partition table entry. This has a maximum size of 32Mb, which is indicated by the system ID of 4. This must set the logical DOS block device as starting at the next cylinder and head boundary. The 05h system ID byte in the second partition entry maps out the space allocated to the next extended volume. The starting cyl/sec/head in the partition entry with ID of 05H is the location of the next extended start-up record of the next extended volume.

Note 8 Logical block device D:. Logical DOS devices always begin with a DOS start-up record as does the primary DOS partition.

Note 9 Extended start-up record for logical block device E:.

Note 10 Partition table entry for logical block device E:. This logical DOS block device is less than 16Mb, as indicated by the system ID of 01h. The entry with system ID of 05H maps out the space allocated to the next extended volume.

Note 11 The system ID byte of 06H indicates a reserved area. These areas are not limited to 32Mb and may not be used in OS/2. This would have a block device letter of F: in a system that would recognize system ID 06h. Note also that a pointer exists to the next extended volume.

Note 12 Reserved

Note 13 Partition table entry for final DOS logical block device. Note the absence of 05h ID byte means that there are no other extended volumes allocated in the extended DOS partition. This would have a block device letter of G: if the previous logical block device was recognized. Otherwise it would be F:.

Partition Table for Master Start-up Record

Offs	Purpose	Head	Sector	Cylinder
1BE	Partition 1 begin	boot ind	H	S	CYL
1C2	Partition 1 end	syst ind	H	S	CYL
1C6	Partition 1 rel sect	Low word	High word
1CA	Partition 1 # sects	Low word	High word
1CE	Partition 2 begin	boot ind	H	S	CYL
1D2	Partition 2 end	syst ind	H	S	CYL
1D6	Partition 2 rel sect	Low word	High word
1DA	Partition 2 # sects	Low word	High word
1DE	Partition 3 begin	boot ind	H	S	CYL
1E2	Partition 3 end	syst ind	H	S	CYL
1E6	Partition 3 rel sect	Low word	High word
1EA	Partition 3 # sects	Low word	High word
1EE	Partition 4 begin	boot ind	H	S	CYL
1F2	Partition 4 end	syst ind	H	S	CYL
1F6	Partition 4 rel sect	Low word	High word
1FA	Partition 4 # sects	Low word	High word
1FE	Signature	

BPB and Get Device Parameters for Extended Volumes

For purposes of the BIOS Parameter Block (BPB) and Get Device Parameters (Generic IOCtl), an extended volume appears to the system as a virtual physical fixed disk. The extended start-up record will correspond to the master start-up record of a real fixed disk and the logical block device will correspond to the primary DOS partition.

This means that the BPB of the logical DOS block device of the extended volume will describe the environment in the extended volume; this consists of the extended start-up record and the logical block device. The meaning of the fields will be consistent to the meaning of the fields for the primary DOS partition; they relate to the entire physical disk, the primary DOS partition, and the master start-up record. For example, the number of hidden sectors will be the distance from the beginning of the extended start-up record (of the extended volume in question) to the start of the logical DOS block device (the DOS start-up record). The number of sectors field will describe only the logical block device just as it normally only describes the primary DOS partition.

Category 8 Generic IOCtl Commands

The philosophy described above also applies to the disk generic IOCtl commands. For any logical block device of an associated extended volume; physical cylinder, head, sector I/O is mapped to within the extended volume. Cylinder 0, head 0, sector 1 is mapped to the extended start-up record. An error condition will be generated for any attempt to do C,H,S I/O beyond the size of the extended volume in question.

Category 9 Generic IOCtl Commands

Category 9 generic IOCtl commands are used to access the entire physical fixed disk without consideration of logical volumes. Physical cylinder, head, sector begin at the start of the physical drive instead of at the beginning of an extended volume. "Get physical device parameters" describes the entire physical device.

EXTDSKDD.SYS Device

This installable device driver allows you to access and use a disk device by referencing a logical drive letter. The format of the CONFIG.SYS DEVICE statement is:

```
DEVICE=EXTDSKDD.SYS /D:ddd[T:ttt][/S:ss]
[/H:hh][/C][/N][/F:f]
```

These parameters are defined as follows:

/D:ddd specifies the physical drive number. A physical drive has the value 0 through 255. A value of 0 specifies the first physical diskette drive and is referenced as drive A from the OS/2 command line.

The value 1 specifies the second physical diskette drive.

The value 2 specifies the third physical diskette drive (which must be external).

A fixed disk drive can be partitioned into many different logical drives using the FDISK utility. Assigning a new drive letter to the physical fixed disk in this environment is not meaningful and is not supported.

/T:ttt specifies the number of tracks per side (1-999). The default is 80 tracks per side.

/S:ss specifies the number of sectors per track (1-99). The default is nine sectors per track.

/H:hh is the maximum number of heads (1-99). The default number is two heads.

/C specifies that Changeline support is required. This is meaningful only on machines that support diskette Changeline, such as the IBM Personal Computer AT and PS/2.

/F:f specifies the device type (form factor). Choose from the list below. The default is 2.

Value	***Device***
0	160/180 KB
0	320/360 KB
1	1.2 MB
2	720 KB
7	1.44 MB

The general rules for drive letter assignment are discussed following these few simple examples. In the first three examples it is assumed that there are no extended DOS partitions on the IBM Personal Computer AT fixed disk.

Example 1 - To set up a logical drive (D) for a 720 KB external diskette drive on a IBM Personal Computer AT (one internal diskette drive and one fixed disk), use the following command:

```
DEVICE=EXTDSKDD.SYS /D:2
```

Example 2 - To be able to copy from a 720 KB external diskette drive to the same drive, put the same command in the CONFIG.SYS file twice, which (for an IBM Personal Computer AT) assigns the logical drive letters D and E to the drive.

```
DEVICE=EXTDSKDD.SYS /D:2
DEVICE=EXTDSKDD.SYS /D:2
```

Example 3 - You can use EXTDSKDD.SYS to copy from an internal drive to the same internal drive. Assume you have an IBM Personal Computer AT with a 1.2Mb drive as the first diskette drive and a 320/360 KB drive as the second physical diskette drive and a fixed disk. The CONFIG.SYS command would be:

```
DEVICE=EXTDSKDD.SYS /D:0 /T:80 /S:15 /H:2 /C /F:1
```

This assigns the logical drive letter D to the first diskette drive. It can now be referenced as A and D. The command

```
A>copy file1 d:
```

copies "file1" from one diskette to another diskette using the 1.2Mb drive only. OS/2 prompts you to insert the diskette for the appropriate logical drive.

Example 4 - If in the previous example FDISK had been used to set up an extended DOS partition on the Personal Computer AT fixed disk, the same DEVICE= statement in CONFIG.SYS would result in the logical drive letter E (instead of D) being assigned to the first diskette drive. It can now be referenced as A and E. The command

```
A>copy file1 e:
```

copies "file1" from one diskette to another diskette using the 1.2Mb drive only. OS/2 prompts you to insert the diskette for the appropriate logical drive.

General Rules For Drive Letters: The first physical internal diskette drive is assigned A. The second internal diskette drive is assigned B. The letters beginning with C are assigned in the order devices (or device drivers) are encountered. The existence of internal physical devices (diskettes and fixed disks) is checked first including partitions on the fixed disks; then the CONFIG.SYS file is checked for device drivers. For OS/2 to recognize an external physical device the CONFIG.SYS file must have the correct device driver information.

The drive letter B automatically is used, even if there is only one physical diskette drive, that is, on machines with only one diskette drive, there are two logical diskette drives A and B. In this case, the parameter /D:1 is an error. The first fixed disk, or the first block device driver, cannot have a drive letter assigned lower than C. On each fixed disk a drive letter is assigned for each primary and extended DOS partition.

For machines with an external drive, if the external device driver is loaded twice, where /D:dd is the same, it generates two logical drives for the one physical drive. This provides the ability to transfer data from one diskette to another in that same drive.

The same concept can also be applied to internal drives. In this case, OS/2 automatically loads a disk device driver for the drive at setup time. By including a DEVICE=EXTDSKDD.SYS in the CONFIG.SYS file for the same drive, two drive letters will be associated with the same drive. The command

```
DEVICE=EXTDSKDD.SYS /D:0
```

in the CONFIG.SYS file at start-up time causes OS/2 to load another diskette driver for the first diskette drive. As described above, the drive letter depends on the number of diskette drives and the number

of fixed disks in the machine. For a machine with two diskette drives and a fixed disk that does not have any extended DOS partitions, the logical drive letter for the first diskette is drive D. With this setup you can copy files from the first physical diskette drive to the first logical diskette drive by referencing them as A and D.

The following table describes the logical drive letter assigned to the external device driver for certain machine configurations and values of /D.

Note: More than one external device driver can be installed at the same time even though this table shows one external device driver. The existence of any VDISKs will not affect the drive letter assignments described below if the DEVICE=VDISK.SYS commands are after the DEVICE=EXTDSKDD.SYS commands in the CONFIG.SYS file.

Internal diskette drives	Internal fixed disk drives *	External drives attached?	Physical drive number (/D:ddd)	Logical drive letter assigned
1	0	No	0	C:
1	0	No	1	error
1	1	No	0	D:
1	1	No	1	error
1	1	No	128	error
1	2	No	0	E:
1	2	No	1	error
1	2	No	128	error
1	0	Yes	0	C: **
1	0	Yes	1	error
1	0	Yes	2	C:

Internal diskette drives	Internal fixed disk drives *	External drives attached?	Physical drive number (/D:ddd)	Logical drive letter assigned
1	1	Yes	0	D:
1	1	Yes	1	error
1	1	Yes	2	D:
1	1	Yes	128	error
1	2	Yes	0	E:
1	2	Yes	1	error
1	2	Yes	2	E:
1	2	Yes	128	error
1	2	Yes	129	error
2	0	No	0	C:
2	0	No	1	C:
2	0	No	2	error
2	1	No	0	D:
2	1	No	1	D:
2	1	No	2	error
2	1	No	128	error
2	2	No	0	E:
2	2	No	1	E:
2	2	No	2	error
2	2	No	128	error
2	2	No	129	error
2	0	No	0	C: **

Internal diskette drives	Internal fixed disk drives *	External drives attached?	Physical drive number (/D:ddd)	Logical drive letter assigned
2	0	Yes	1	C: **
2	0	Yes	2	C:
2	1	Yes	0	D: **
2	1	Yes	1	D: **
2	1	Yes	2	D:
2	1	Yes	128	error
2	2	Yes	0	E: **
2	2	Yes	1	E: **
2	2	Yes	2	E:
2	2	Yes	128	error
2	2	Yes	129	error

* These values assume that there are no extended DOS partitions on the fixed disk(s).

** The external drive is not recognized.

Note that a physical drive number above 127 is an error.

Printer Device Driver

The printer device driver runs in a multitasking, dual-mode environment.

The following functions are supported:

- Initialize Printer Device Driver
- Print Character
- Return Printer Device Driver Status
- Open Device
- Close Device
- Deinstall
- Generic IOCtl functions
  - Set/Return Frame Control
  - Set/Return Infinite Retry
  - Initialize Printer Port
  - Return Printer Port Status
  - Register/Query Monitor Support
  - Activate Code Page/Font
  - Query Code Page/Font
  - Verify Code Page/Font

When the user asks for printer status with the generic IOCtl request, the following information is returned:

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

The printer device driver provides character device monitor support on a per device basis. The printer device driver supports the registration of two monitor chains. The first chain (INDEX=1) is normally considered to be the data chain and is used by the printer device driver to send character data to a monitor. The second chain (INDEX=2) is normally considered to be the code page chain, but it is used by the printer device driver only to receive the results of a code page request. The code page request sent to the monitor was performed using the first chain (INDEX=1).

The printer spooler is an example of a monitor that registers itself on both the data chain and the code page result chain. The input buffer specified when the spooler registers INDEX=2 is never used by the spooler. All data and code page requests from the printer device driver to the spooler on one chain are sent to insure that the spooler processed the requests in order. The second chain was implemented to enhance processing. Processes that issue code page requests are blocked until they receive an indication that their request is valid. The second chain allows a monitor to respond to the printer device driver quickly and efficiently.

Therefore, printer device driver monitors must be designed with extreme care and thought must be given to their position in the chain when other monitors are involved. The following items list some of the possibilities:

Notes:

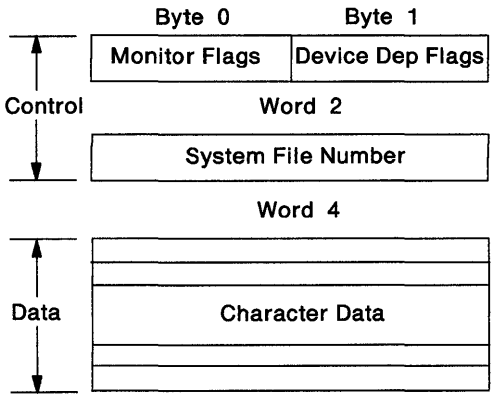
1. If a monitor wishes only to process character data and it is the only monitor in the chain, or it is registered to be in a position to process the data after the spooler, it only needs to register on the data chain (INDEX=1). This monitor would never see the code page requests from the spooler, because the spooler sends these requests out on the code page result chain. If the spooler wasn't part of the chain, then the printer device driver would never issue the code page requests.

2. If a monitor wishes to process both the character data and the code page requests and it is the only monitor in the chain, it must register itself on both the data and code page result chains. It must also expect to receive both the data and code page requests on the data chain, it must respond to the code page requests on the result chain as quickly as possible.

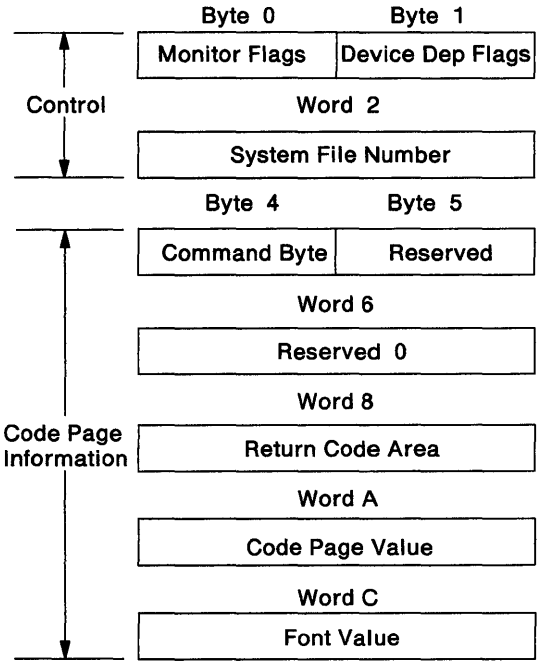
3. If a monitor wishes to process both the character data and the code page requests and its position in the chain is after the printer spooler, it must expect to receive the character data from the spooler on its data chain (INDEX=1) and the code page requests from the spooler on the code page chain (INDEX=2). A monitor in this situation would not be able to easily synchronize the code page requests with the data requests; the spooler will pass the code page results along the result chain before all the data has been spooled and released.

4. If a monitor wishes to process both the character data and the code page requests and its position in the chain is before the printer spooler, it registers itself on the data chain (INDEX=1). The spooler will register itself on both chains. All data and code page requests will be sent to the first monitor on the data chain. This monitor must pass on to the spooler all the information it receives, in the order received.

The printer device driver passes the character information to the monitors in data packets. The character information in the data packets is preceded by control information. The maximum size of the data packet which consists of both the control information and the character data is 134 bytes. This value is used to calculate the size of the monitor input/output buffers required for the DosMonReg call.



The printer device driver passes the code page information to the monitors in data packets. The code page information in the data packets is preceded by control information.



The printer device driver is in INIT MODE when its strategy routine is called with a request packet containing the INIT command. The initialization code runs in the OS/2 mode at the application privilege level with I/O privilege.

The printer device driver will handle print requests received at its strategy entry point from the file system in the form of request packets as well as INT17H software interrupt requests received from the DOS mode while in user mode.

The printer device driver has three printer specific IOCtl commands to support the code page and font switching provided in OS/2. All of the actual code page and font switching function for printers is provided by the code page switcher spooler. When the spooler is started, it checks to see if code page support is required. If it is, the spooler will cause the code page switcher support to be loaded and initialized. In order to support these IOCtls, there are Font Monitor Buffer Command/responses in the monitor interface to the spooler.

The functions provided by the IOCtls and Font Monitor Buffer Command/responses are:

- Activate Font
- Query Active Font
- Verify Font

These IOCtls and monitor code page and font buffer command/responses are described in detail in "Printer Device Driver Interfaces" on page 9-159.

Activate Font: Refer to "Printer/Spooler Structure" on page 6-55, when an application within a process issues a DosOpen for a printer (i.e. LPT1, LPT2, LPT3 or PRN). The file system issues an Activate Font IOCtl to the printer device driver to set the active code page and font according the active code page of the process making the open request.

An application within a process may also issue the Activate Font IOCtl to the printer device driver by using the handle returned from the DosOpen at any time. This allows the application to change the active code page and font in the middle of its print job.

When the printer device driver receives the Activate Font IOCtl, if the spooler or another monitor is registered, the printer device driver will use the Font Monitor Buffer command to send the Activate Font command to the registered monitor in its monitor input buffer. If the spooler, or another monitor, is not registered, the printer device driver will return the appropriate return code to the caller of the Activate Font IOCtl.

Query Active Font: When the printer device driver receives the Query Active Font IOCtl, if the spooler or another monitor is registered, the printer device driver will use the Font Monitor Buffer command to send the Query Active Font command to the registered monitor in its monitor input buffer. The monitor will return the active code page and font information to the printer device driver in its monitor output buffer. The printer device driver then returns the active code page and font information to the caller of the Query Active Font IOCtl.

If the spooler, or another monitor, is not registered, the printer device driver will return the appropriate return code to the caller of the Query Active Font IOCtl.

Verify Font: When the printer device driver receives the Verify Font IOCtl, if the spooler or another monitor is registered, the printer device driver will use the Font Monitor Buffer command to send the Verify Font command to the registered monitor in its monitor input buffer. The monitor will return whether the specified code page and font is valid to the printer device driver in its monitor output buffer. The printer device driver then returns the return code to the caller of the Verify Font IOCtl.

If the spooler, or another monitor, is not registered the printer device driver will return the appropriate return code to the caller of the Verify Font IOCtl.

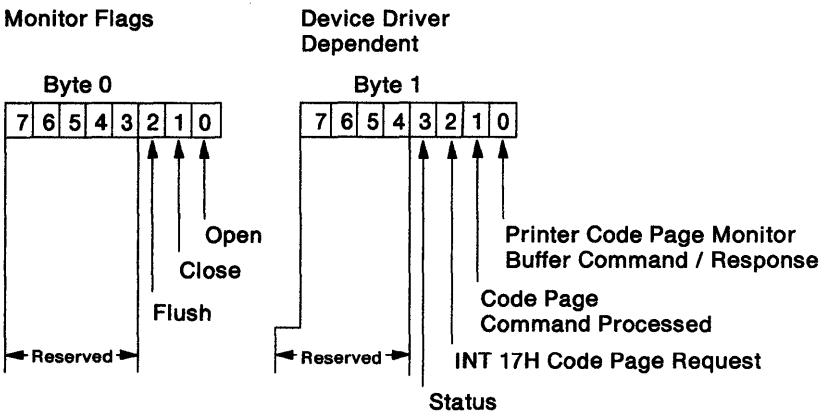
Printer Device Driver Interfaces

The interfaces required by the printer device driver for code page and font switching are:

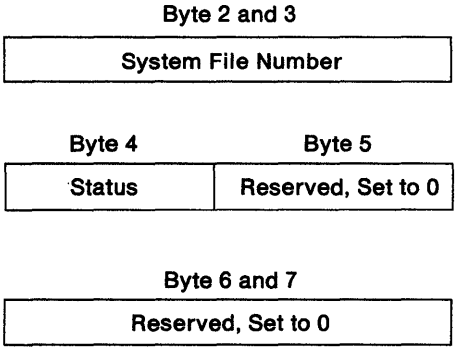
- Activate Font IOCtl - Category 5 Function 48h
- Query Active Font IOCtl - Category 5 Function 69h
- Verify Font IOCtl - Category 5 Function 6Ah
- Font Monitor Buffer Commands
  - Activate Font
  - Query Active Font
  - Verify Font

Font Monitor Buffer Commands

Printer Monitor Record: Each monitor record can be of variable length. However, there must be a word of flags at the beginning of each monitor record. The flags are defined as follows:



When the Status bit is set to 1 (byte 1, bit 3), this indicates that the monitor buffer is a status packet from the printer device driver. In this case, the next six bytes are:



Byte 2 & 3 System File Number WORD

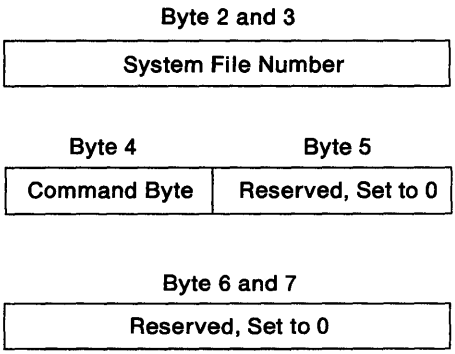
Byte 4 Status byte which indicates the type of printer error. This will be the same 8 bits returned as the status field in a request header.

Byte 5 Reserved, and must be set to zero (0).

Byte 6 & 7 Reserved WORD, must be set to zero (0).

When the Font Monitor Buffer command bit is set to 1 (byte 1, bit 0), this indicates that the monitor buffer is a Font Monitor Buffer

command or a response to a previous Font Monitor Buffer command. In this case, the next six bytes are:



Byte 2 & 3 System File Number

Byte 4 Font Monitor Buffer Command Byte which indicates the type of command or response.

Byte 5 Reserved, and must be set to 0.

Byte 6 & 7 Reserved WORD, must be set to 0.

When the Code Page Command Processed bit is set to 1 (byte 1, bit 1), the Font Monitor Buffer command has been processed by the spooler. The printer device driver sends Font Monitor Buffer commands on monitor chain Index=1. A monitor which services the Font Monitor Buffer command will use MonWrite to place the Font Monitor Buffer Response on Monitor chain Index=2. Data to be printed continues to flow on monitor chain Index=1. Monitor chain Index=2 is used only for the Font Monitor Buffer responses so that the printer device driver does not block a program issuing a code page and font IOCtl request when print data monitor buffers are already queued ahead of the IOCtl request.

When the INT17H Code Page Request bit is set to 1 (byte 1, bit 2), the Activate Code Page Request was issued automatically by the printer device driver on behalf of INT17H processing and the application is not waiting for a response.

The valid Font Monitor Buffer commands, along with the remainder of the buffer contents for the responses, are as follows:

Byte 4 = 01H Activate Font

Command Data, starting at byte 8:

08	WORD	Return Code
0A	WORD	Code Page
0C	WORD	Font Id

Return Code A WORD value starting at byte 8, and includes the following values:

- Successful completion
- Code page and font switching not active
- Unable to access specified font file
- Unable to locate or activate specified code page
- Unable to locate or activate specified font.

Code Page The value of the code page to make the currently active code page.

0000H If both the code page value and font ID are specified as 0, set printer to hardware default code page and font.

0001H-FFFFH Valid code page numbers.

FontId The ID value of the font to make currently active.

0000H If **both** the code page value and Font Id are specified as zero (0), set printer to hardware default code page and font.

If only the font Id is specified as 0, any font within the specified code page is acceptable.

0001H-FFFFH Valid font ID numbers, font types defined by the font file definitions. .*

Remarks The Font Monitor Buffer command is passed on monitor chain index=1 by the printer device driver to the spooler. The Font Monitor Buffer response is returned by the spooler on monitor chain index=2 to the printer device driver.

Byte 4 = 02H Query Active Font

There is no additional command data.

The response is as follows:

08	WORD	Return Code
0A	WORD	Code Page
0C	WORD	Font Id

Return Code A WORD value starting at byte 8, and includes the following values:

- Successful completion
- Code page and font switching not active.

Code Page The value of the currently active code page.

Valid numbers are between 0 and 65535.

A value of 0 for **both** the code page and font ID indicates that the hardware default code page and font is active.

Font Id The ID value of the currently active font.

Valid numbers are between 0 and 65535.

Remarks The Font Monitor Buffer command is passed on monitor chain index = 1 by the printer device driver to the spooler. The Font Monitor Buffer response is returned by the spooler on monitor chain index = 2 to the printer device driver.

Byte 4 = 03H Verify Font

Command Data, starting at byte 8:

Return Code A WORD value starting at byte 8, and includes the following values:

- Code page and font valid
- Code page not valid
- Font not valid

- Code page and font switching not active
- Unable to access specified font file.

Code Page The value of the code page to validate.

Values may be 0 to 65535.

FontId The Font ID to validate.

Values may be 0 to 65535. A value of 0 indicates that any font within the specified code page is acceptable.

Remarks The Font Monitor Buffer command is passed on monitor chain index = 1 by the printer device driver to the spooler. The Font Monitor Buffer response is returned by the spooler on monitor chain index = 2 to the printer device driver. Because the Activate Font may also contain data to be printed, it may also be returned by the spooler on monitor chain index = 1.

Chapter 10. Functions and Utilities for Problem Determination

Design Elements

Reliability Functions

Semaphores

Semaphore mechanisms sequence the updating or processing of data by asynchronous operations.

Diskette processing

OS/2 maintains an awareness of the volume label associated with each open file and an awareness of the volume label on the current diskette in each drive. You are instructed to insert the correct diskette whenever I/O is requested for a diskette which is not the current one. After you respond, OS/2 reads and compares the volume label to verify that the correct volume has been inserted. OS/2 then performs the requested I/O. This feature is supported for file I/O using file handles.

File write-through

Applications have the ability to know that a successful file write operation has occurred in order to sequence file updates to minimize the impact of a system failure. This can be done with synchronous file I/O to synchronous files. The VERIFY command is available for validating file updates.

Availability Functions

The following items help maximize system availability. They also can be used to improve reliability.

Termination exit capability — Termination exits are provided.

Error recovery — An application can attempt recovery from any error it detects. Upon detection of an error, the application can attempt to: retry, ignore, correct, or abort.

Localization of Damage — Termination is done on a process basis. When an error occurs, and error recovery cannot be accomplished, only the affected process(es) are terminated.

Serviceability Functions

The following functions are provided for problem determination and isolation in a multitasking environment:

Create Dump Diskette Utility — Creates a diskette to be used with the stand-alone dump facility. This utility should be executed immediately following system installation.

Note: This procedure (CREATEDD) is required only ONCE for each dump. It does not need to be done on every re-installation/re-start.

System Trace Facility — Allows important system events to be recorded.

Stand-alone Dump Facility — Initiated by pressing Ctrl/Alt/NumLock twice.

Note: This procedure will require a system restart (re-boot) after the dump is complete.

System Trace Facility

The OS/2 System Trace Facility helps support personnel in diagnosing system problems. It provides an internal system interface that allows the placing of variable length data in a variable size circular trace buffer. This provides a historical trail of system events that could be useful in isolating the cause of a system failure.

Using System Trace

TRACEBUF and TRACE commands in CONFIG.SYS.

There are two CONFIG.SYS commands used to indicate that system tracing is desired. The format of the commands follows:

```
TRACEBUF = y
```

where *y* is an integer that represents the size of the Circular Trace Buffer in 1K increments.

- If the 'TRACEBUF' command is invalid, the command is ignored and a 'TRACEBUF' will be allocated only if a 'TRACE' command is specified.
- If this facility receives errors in trying to allocate the trace buffer and associated data, an error message will be reported and tracing will never be enabled.
- If multiple 'TRACEBUF' commands are given, the last one supersedes all previous occurrences.
- If a 'TRACE' command is in the CONFIG.SYS file, but no valid 'TRACEBUF' command is present, a 4K buffer is allocated.
- If a 'TRACEBUF' command is in the CONFIG.SYS file, but no 'TRACE' command is present, the buffer exists (if it could be allocated), but no trace events will be enabled during initialization.
- If neither 'TRACEBUF' nor 'TRACE' appears in CONFIG.SYS, no trace buffer will be allocated and tracing will never be enabled.

```
TRACE =(ON ¦ OFF) [x [,x ...]]
```

Where *x* is an integer that indicates whether an event is to be enabled (ON) or disabled (OFF) for system tracing, multiple events can be

individually selected by listing one after the other. The values of the numbers are between 0 and 255, and are referred to as major event codes. Examples of events associated with different major event codes are tasking events or hardware interrupts.

- An invalid 'TRACE' command results in error message(s) and is ignored. If the command is invalid, the buffer is allocated only if either 'TRACE' or 'TRACEBUF' is specified.
- If no parameters are listed after the ON/OFF parameter, then ON indicates all trace calls encountered are traced and OFF means no tracing is done.
- Multiple 'TRACE' commands are allowed in the CONFIG.SYS file.
- If an event code is referenced in multiple 'TRACE' commands, the last reference supersedes all others.

TRACE as an OS/2 Command Utility

The OS/2 command utility 'TRACE' provides dynamic enabling/disabling of tracing by the user. This utility is available only in the OS/2 mode. The format of the command is :

```
[d:] [path] TRACE (ON ¦ OFF)  [x [,x ...]]
```

where *x* is a decimal integer between 0 and 255 inclusive, corresponding to a major trace event code that is to be traced (ON) or not traced (OFF).

- If no parameters are listed after the ON/OFF parameter, then ON indicates all trace calls encountered are traced and OFF means no tracing is done.
- If no trace buffer was allocated during the processing of the CONFIG.SYS file at initialization time, an error message will result and no tracing is provided by the system.
- If a number is outside of the given range, an error message is generated for that number and processing of the remainder of the command continues.
- If the ON/OFF parameter is invalid, or all of the major event codes are invalid, a message is generated and the entire command is ignored.

Considerations/Limitations

The trace buffer is circular and wraps. In some cases, you may lose useful information if trace is not disabled (with TRACE OFF) immediately after the problem has occurred.

Stand-Alone Dump Facility

The Stand-Alone Dump facility aids in providing problem determination services for OS/2. The OS/2 Stand-Alone Dump facility is used to dump to diskettes all physical memory, including the 640K - 1M address space. Memory between 640K and 1M contains information belonging to I/O adapters that have shared memory (display, smart communications adapters, and others).

The major portion of the Stand-Alone Dump functions independently of OS/2, although a portion of the Dump facility is part of the fixed and memory resident portion of OS/2. The purpose of the memory resident code is to stabilize the system hardware and initiate the start of the Stand-alone Dump diskette.

A keystroke sequence (Ctrl + Alt + NumLock pressed twice) is entered to invoke the stand-alone dump. That keystroke sequence is detected by the keyboard device driver and control is passed to memory resident stand-alone dump code. That code sets up the hardware so a start-up of a diskette can be done. It then starts a stand-alone dump diskette to initiate the actual dumping of physical memory. The stand-alone-dump diskette is created with the Create Dump Diskette Utility. The stand-alone dump will be analyzed by an IBM Customer Representative.

Initiating a Dump

Procedure used to start a Dump

To initiate a dump, press the NumLock key twice in a row while holding the Ctrl + Alt keys down. The user should exercise caution when initiating a dump because system activities in progress will stop without cleanup. The Ctrl + Alt + NumLock keys may be used in either the OS/2 mode or the DOS mode. A prompt will instruct you to insert the stand-alone dump diskette in drive A. The dump code will verify that a CREATEDD diskette has been inserted in drive A. This is done to avoid an accidental start on a fixed disk system by placing an incorrect diskette in drive A.

From this point, all physical memory will be dumped to diskette(s). If additional diskettes are needed after the initial stand-alone dump diskette, then formatted diskettes can be used. Each time you are prompted to insert another diskette, you may end the memory dump by re-inserting the first dump diskette.

If additional diskettes are required, a header record will be placed on each additional diskette. This header record also contains the summary information available at the time this diskette is loaded for dumping. After the dump has completed, a summary record will be written on the first stand-alone dump diskette. The summary record will contain, at a minimum, information pertaining to the range of physical memory dumped, the number of diskettes used in the dump, and the physical memory range that was placed on each diskette. Once the dump is complete, you need to re-start OS/2 in order to use OS/2 again.

Trace Formatter Utility

The Trace Formatter Utility is a service aid that can be used in conjunction with the system trace facility to debug problems in the OS/2 system. The facility captures the current contents of the system trace buffer, analyzes each trace record, and displays the formatted data on the standard output device. The standard output device can be the display, a printer or a file.

The format of the command is:

```
[d:] [path] TRACEFMT
```

[d:] [path] is the optional drive and path where the system can find the TRACEFMT utility.

To use the trace formatter, a trace buffer must have been allocated at IPL time. To allocate a trace buffer, you must include either the TRACEBUF or TRACE command in CONFIG.SYS. Tracing is not required to be active (that is, TRACE = ON) at the time the TRACEFMT command is issued.

The trace formatter can be invoked as many times as required to diagnose the problem.

The formatted trace records are displayed in reverse time-stamp order (that is, newest records first). Each formatted trace record consists of header information and optional data. The header information contains:

- The title of the trace event. The title is a string describing the event. Each major/minor event code combination is converted to a unique title.
- For events that have a pre-invocation and post-invocation code, a string is displayed indicating whether the trace record is the pre-invocation or post-invocation of the event.

- The issuing process ID in the form "Issuing Process ID = xxxx"
- Two strings that represent the trace record flag word as follows:
  - "Kernel Call" or "Dynlink Call" to indicate the type of call invocation
  - "OS/2 Mode" or "DOS Mode" to indicate the mode of the calling process
- The time stamp of the event in the form "Time Stamp = ss.hh, where "ss" is seconds and "hh" is hundredths of seconds. If the time stamp is the same as the previous trace record, the time will not be displayed. Instead, the string "Time Stamp = ..." is displayed.

Following the header information fields, the optional event specific data is displayed. If unrecognized trace records have been added to the trace buffer, the trace formatter will display "Unrecognized Trace Event" as the event title, followed by the rest of the header data. It will then display the major and minor code contained in the trace record.

An example formatted trace output appears below:

```
DosHoldSignal Post-Invocation
  Issuing Process ID=0006  OS/2 Mode Kernel Call Time Stamp=....
  Return Code=0000

DosHoldSignal Pre-Invocation
  Issuing Process ID=0006  OS/2 Mode Kernel Call Time Stamp=....
  Action Code=0000

DosGetShrSeg Post-Invocation
  Issuing Process ID=0006  OS/2 Mode Kernel Call Time Stamp=....
  Selector=0307  Return Code=0000

DosGetShrSeg Pre-Invocation
  Issuing Process ID=0006  OS/2 Mode Kernel Call Time Stamp=23.44
  Name=\SHAREMEM\SMG\SGTITLE
```

Create Dump Diskette Utility

This utility is used to create the initial, start (boot) diskette that contains the files necessary to perform the stand-alone dump. The diskette is also made to appear full to prevent accidentally placing data other than from the dump on the diskette. If additional dump diskettes are needed for a given dump, use diskettes that were formatted with the FAT based OS/2 FORMAT utility.

You can only invoke this utility from the OS/2 mode of OS/2.

Using the Create Dump Diskette Utility

To invoke the Create Dump Diskette utility, enter the following:

```
[d:] [path] CREATEDD x
```

Where:

[d:] [path] - is the optional drive and path where the CREATEDD utility can be found by the system.

x - is the destination drive that contains the diskette to be used as the dump diskette.

If no parameter or an an invalid parameter is specified, the utility ends and no dump diskette is created.

The CREATEDD utility calls the FORMAT utility. Therefore, FORMAT must reside in the Default Directory or the Path must be set up to find the FORMAT utility.

Chapter 11. Country Support Considerations

Introduction

Country Support for OS/2 includes these features:

- Country dependent information
- Country APIs
- National keyboard layouts
- Configuration commands
- Translated system message files

Country Dependent Information

Country information is available in OS/2 for all the countries and corresponding country codes listed below:

Country	Code
United States	001
Canada	002
Latin America	003
Netherlands	031
Belgium	032
France	033
Spain	034
Italy	039
Switzerland	041
United Kingdom	044
Denmark	045
Sweden	046
Norway	047

Country	Code
Germany	049
Australia	061
Japan	081
Korea	082
People's Republic of China	086
Taiwan	088
Asia	099
Portugal	351
Finland	358
Arabic	785
Hebrew	972

For each country code there is a corresponding set of country dependent information contained in the COUNTRY.SYS file that includes:

- Time and date format, currency symbol and decimal separator information
- Lower to upper case mapping table for ASCII characters
- Collating sequence table for ASCII character string sorting by SORT utility
- DBCS environmental vector table for double-byte character sets.

The set of country dependent information that is used by OS/2 is determined by the country code the COUNTRY command is set to in CONFIG.SYS. This system country code is always the same for all application sessions. Country information retrieval is based on the country code and code page of the calling process or a selected country code and selected code page.

Applications can request country information for the system country code used by OS/2 or a specific country code using the Country API calls:

DosGetCtryInfo - Get the time, date and other format information for the current country code or a selected country code.

DosCaseMap - A variable length string of ASCII characters is case mapped from lower to upper case and returned.

DosGetCollate - Get the collate sequence table for sorting

DosGetDBCSEv - Get the DBCS environment vector binary value ranges of valid lead bytes for double-byte characters.

Please refer to *Technical Reference, Vol. 2* for more detailed information about function calls and the format of the information returned.

National Keyboard Layouts

System keyboard layouts are available for different countries. The following table lists these different countries and the two-letter code that is used to select the desired keyboard layout with the KEYB utility:

Keyboard	Code
Belgium	BE
Canadian-French	CF
Denmark	DK
Finland	SU
France	FR
Germany	GR
Italy	IT
Latin America	LA
Netherlands	NL
Norway	NO
Portugal	PO
Spain	SP
Sweden	SV
Swiss-French	SF
Swiss-German	SG
United Kingdom	UK
United States	US

The system keyboard layout is selected by the utility command KEYB and the same layout is used for all application sessions. Multiple KEYB commands can be selected by the user in a given session. The last keyboard layout selected determines the current keyboard layout in the system. The United States keyboard layout is the default layout.

Note: This two-letter code is also used in the CONFIG.SYS DEVINFO= command.

Code Page Configuration

Code page configuration of the system is necessary to be able to successfully switch between two code pages at run time. The following set of commands must be set up correctly in CONFIG.SYS for this purpose:

Command	*Purpose*
CODEPAGE	Specify one or two code page identifiers the system is to set up to use.
COUNTRY	Specify the country code and a fully specified file name. The file contains a set of country information characters encoded according to a code page based on ASCII. The system defaults to the COUNTRY.SYS file in the root directory of the start drive if no file is specified.
DEVINFO	Specify the keyboard layout selection and a fully specified file name. The file contains a keyboard layout table for translating keystrokes into characters encoded according to a code page based on ASCII. The system defaults to the KEYBOARD.DCP file in the root directory of the boot drive if no file is specified.
DEVINFO	Specify for the display device a fully specified file name. The file contains a video font table for displaying characters encoded according to a code page based on ASCII. The system does not have a default file name but the file VIOTBL.DCP is provided.
DEVINFO	Specify for the printer device a fully specified file name. The file contains a printer font table for printing characters encoded according to a code page based on ASCII. The system does not have a default file name but the files 4201.DCP and 5201.DCP are provided.

Incorrect, partial, or mismatched set-up of commands for code page selections, country code, keyboard layout, display, and printer may cause ineffective switching between code pages at run time. See the User Reference for the description and syntax of each command and the CHCP change code page command.

Utility and Configuration Command Support

The configuration command COUNTRY has the format COUNTRY=xxx where xxx is the country code for the country dependent information. This command causes the current country code to change from the system default (United States=001) to xxx.

The various countries supported and their associated country codes are discussed earlier in this chapter.

The CONFIG.SYS RUN= command can be used with the KEYB utility to start the system with a system keyboard layout other than the U.S. default. The KEYB [*yy*] utility is used to change the system keyboard layout.

The OS/2 System Installation allows system installation for a selected country code and keyboard layout.

Appendix A. The Linker

This appendix contains the following subjects:

- Converting object files to executable code
- Linker options
- Linker error messages
- Module definition file statements

Note: The OS/2 Linker does not support the use of the APPEND command and will not find run time libraries in APPEND directories.

Converting Object Files to Executable Code

To convert object code files to executable code, the OS/2 Linker requires the following syntax:

```
LINK object-list,[run-file],[map-file],[library-list],
[definition-file],[/options list];
```

The preceding syntax has the following meaning:

object-list	A list of object files to be linked together. Separate filenames with plus (+) signs or blanks. The default extension is .OBJ.
[run-file]	The name of the file to receive the executable output. The default filename is the name of the first listed object file. The default extension is .EXE.
[map-file]	The name of the file to receive the map listing.
[library-list]	A list of libraries for LINK to search. Separate list items with plus (+) signs or blanks. The default extension is .LIB.
[definition-file]	An optional module definition file.
[/options list]	An optional list of linker parameters.

The commas (,) shown in the command line syntax are required if all the input fields are specified on a single line. If the command line does not end with a semicolon (;), the linker will prompt for any

remaining input fields not given on the command line or in a response file.

Note: For further information on linking, see Chapters 7 and 8 in the OS/2 Programmer's Guide. Those chapters explain:

- How to link dynamic link libraries
- How to use the linker by:
  - Responding to prompts
  - Typing on the command line
  - Creating an Automatic Response File (ARF).

About LINK Options

LINK can be used to link programs written with the IBM languages in either the OS/2 or DOS environments. The overlay capability is for use in the DOS 3.3 environment only.

Using LINK Options

The linker options control the tasks performed by LINK. You may specify an option anywhere before the last comma on the command or response line. Every option must begin with the slash character, even if other options appear before it on the line.

You can abbreviate option names as long as your abbreviations contain enough letters to distinguish the specified option from other options. Minimum abbreviations are listed with the description for each option on the following pages. Link does not recognize spaces between characters, nor does it recognize transpositions (changes in order). The link options are:

Option	Description
/ALIGNMENT	Sets segment alignment factor
/CODEVIEW	Includes debugging information for the CodeView debugger
/CPARMAXALLOC	Changes value of maximum number of reserved paragraphs
/DOSSEG	Forces ordering of segments
/DSALLOCATE	Controls data loading

/EXEPACK	Packs executable files
/FARCALLTRANSLATION	Optimizes intra-segment far calls
/HELP	Writes a list of the available options to the screen
/HIGH	Controls loading the run file
/INFORMATION	Displays information about the linking process
/LINENUMBERS	Copies line numbers to the map file
/MAP	Lists all public symbols in your program
/NODEFAULTLIBRARYSEARCH	Ignores default libraries
/NOFARCALLTRANSLATION	Disables far call translations
/NOGROUPASSOCIATION	Provides compatibility with previous compiler versions
/NOIGNORECASE	Case sensitive
/NOPACKCODE	Disables code segment packing
/OVERLAYINTERRUPT	Sets the overlay
/PACKCODE	Packs code segments
/PAUSE	Pauses to change disks
/SEGMENTS	Sets the maximum number of segments
/STACK	Sets the stack size
/WARNFIXUP	Warns of incorrect offset

Entry of Numeric Parameters

Numeric parameters on the linker options and in the module definition statements can be entered in decimal, hexadecimal, or octal. The format follows the C language conventions.

The C entry conventions are:

Decimal	Any number which begins with anything other than a 0 digit. For example: 1, 65536, 2084, 234
Hexadecimal	A number which must begin with 0x and contain 0 - 9, A - F. For example: 0xFFF, 0x10
Octal	Any number which begins with the digit 0 which contains only the digits 0 - 7. For example: 010, 05000, 0777

This option directs LINK to set the segment alignment factor in the executable file to the number given, which must be a power of 2. The default alignment is 512.

This option is valid only for code linked to run in the OS/2 environment.

Format

/ALIGNMENT:*number*

The minimum abbreviation is **/A**.

Remarks

Aligning a segment means adjusting its address to the next address occurring on a specific boundary based on the alignment factor. The adjusted, or aligned, address will then be evenly divisible by the alignment factor.

The *number* can be a hexadecimal, decimal, or octal number.

/ALIGNMENT tells the linker to adjust the beginning of a segment to the next address boundary which matches the specified alignment factor.

Preparing Files for CodeView
/CODEVIEW

This option directs LINK to include symbolic debugging information for CodeView in the output executable file.

Format

/CODEVIEW

The minimum abbreviation is **/CO.**

Remarks

/CODEVIEW cannot be used with /EXEPACK.

Reserving Paragraph Space /CPARMAXALLOC

This option allows you to change the default value of the MAXALLOC field, which controls the maximum number of paragraphs reserved in storage for your program. A paragraph is defined as the smallest storage unit (16 bytes) addressable by a segment register.

This option is valid only for code linked to run in the DOS 3.3 environment or in the DOS environment of OS/2.

Format

/CPARMAXALLOC:*number*

The minimum abbreviation is **/CP**.

Remarks

The maximum number of paragraphs reserved for a program is determined by the value of the MAXALLOC field at offset 0CH in the EXE header.

The default for the MAXALLOC field is 65535 (decimal), or 64K minus 1. You can reset the default to any number between 1 and 65535 (decimal, octal, or hexadecimal). Changing the number is helpful because:

- Program efficiency is not increased by reserving all available storage.
- You may need to run another program within your program and you need to reserve space for that program.

If the value you specify is less than the computed value of MINALLOC (at offset 0AH), the linker uses the value of MINALLOC instead.

Ordering Segments /DOSSEG

The /DOSSEG option forces segments to be ordered according to the following rules:

1. All segments with a class name ending in CODE.
2. All other segments outside of DGROUP.
3. DGROUP segments in the following order:
   a. Any segments of class BEGDATA. (This class name is reserved for IBM use.)
   b. Any segments not of class BEGDATA, BSS, or STACK.
   c. Segments of class BSS.
   d. Segments of class STACK.

Format

/DOSSEG

The minimum abbreviation is **/DO**.

Remarks

Linking with the standard IBM C runtime libraries automatically enables the /DOSSEG option by way of a comment record in the startup module.

An additional effect of this option is that the linker inserts 16 bytes of NULLs in front of the segment named _TEXT, if present. This is necessary for C runtime library support.

By default, LINK loads all data starting at the low end of the data segment. At run time, LINK sets the DS (data segment) pointer to the lowest possible address to allow the entire data segment to be used.

You can use the /DSALLOCATE option to tell LINK to load all data starting at the high end of the data segment. To do this, set the DS pointer at run time to the lowest data segment address that contains program data.

This option is valid only for code linked to run in the DOS 3.3 environment or in the DOS environment of OS/2.

Format

/DSALLOCATE

The minimum abbreviation is **/DS**.

Remarks

The /DSALLOCATE option is typically used with the /HIGH option to take advantage of unused storage within the data segment. You can reserve any available storage below the area specifically reserved for DGROUP, using the same **DS** pointer.

Packing Executable Files /EXEPACK

This option directs LINK to remove sequences of repeated bytes (typically nulls) and to optimize the load-time relocation table before creating the DOS executable file. This option is valid only for code linked to run in DOS 3.3 or in the DOS mode of OS/2.

OS/2 executable files are always compressed.

Format

/EXEPACK

The minimum abbreviation is **/E**.

Remarks

Executable files linked with this option are usually smaller and load faster than files linked without this option. However, you cannot use symbolic debugging programs with packed files.

The /EXEPACK option does not always save a significant amount of disk space and may sometimes increase file size. Programs that have a large number of load-time relocations (about 500 or more) or long streams of repeated characters are usually shorter if packed.

/EXEPACK cannot be used with /CODEVIEW.

This option directs LINK to optimize intra-segment far calls into the sequence

```
NOP
PUSH CS
CALL NEAR address
```

By default, the linker does not perform this optimization.

This option is valid only for code linked to run in the OS/2 environment.

Format

/FARCALLTRANSLATION

The minimum abbreviation is **/FAR**.

Remarks

In most medium and large model programs, this option will yield significant savings in executable size and load time. However, there is a small chance that the linker, during this optimization, will mistakenly identify a byte with a value of 0x9a as a far call, when in fact it is an assembled constant. Take care when using this option.

Viewing the Options List /HELP

The /HELP option causes LINK to write a list of the available options to the screen. This may be convenient if you need a reminder of the available options. Do not give a file name when using the /HELP option.

Format

/HELP

The minimum abbreviation is **/HE**.

The /HIGH option loads the run file as high as possible in storage without overlaying the transient portion of COMMAND.COM. The COMMAND.COM file occupies the highest area of storage when loaded and the run file as low as possible.

Use the /HIGH option in association with the /DSALLOCATE option.

This option is valid only for code linked to run in the DOS 3.3 environment or in the DOS environment of OS/2.

Format

/HIGH

The minimum abbreviation is **/HI**.

Remarks

The /HIGH option should not be used with programs written in C.

Displaying Information about the Linking Process /INFORMATION

The /INFORMATION option directs the linker to display information about the linking process, including the phase of linking and the full name of each module as it is processed.

Format

/INFORMATION

The minimum abbreviation is **/I**.

Remarks

/INFORMATION is useful for debugging.

Copying Line Numbers to the Map File /LINENUMBERS

The /LINENUMBERS option directs the linker to copy the starting address of each program source line to a map file. The starting address is the address of the first instruction that corresponds to the source line.

Format

/LINENUMBERS

The minimum abbreviation is **/LI**.

Remarks

LINK copies the line number data only if you give a map file name in the LINK command line and only if the given object file has line number information. Line numbering is available in some high level languages.

The Macro Assembler does not copy line number information to the object file. If an object file has no line number information, the linker ignores the /LINENUMBERS option.

If you do not specify a map file in a LINK command, you can still use the /LINENUMBERS option to force the linker to create a map file. Just place the option at or before the **List File** prompt. LINK gives the forced map file the same file name as the first object file specified in the command and gives it the default extension .MAP.

Producing a Public Symbol Map /MAP

The /MAP option causes LINK to produce a listing of all public symbols declared in your program. This list is copied to the map file created by the linker.

Format

/MAP [:*number***]**
The minimum abbreviation is **/M.**

Remarks

The *number* parameter specifies the maximum number of public symbols that the linker can sort in the map file. If you give no number, the limit is 2048. Valid values are 1 through 32767. They can be specified in hex, decimal, or octal.

Specifying a number also causes the public symbols to be sorted by address only and not by name, regardless of the number. If you want to reduce the size of your map files by removing the list sorted by name, link with /MAP followed by a small number, but large enough to accommodate the number of public symbols in your program.

If you do not specify a map file in a LINK command, you can use the /MAP option to force the linker to create a map file. LINK gives the forced map file the same name as the first object file specified in the command and the default extension .MAP.

For a complete description and examples of the listing file format, see "The Map File" in this appendix.

Ignoring Default Libraries /NODEFAULTLIBRARYSEARCH

The /NODEFAULTLIBRARYSEARCH option directs the linker to ignore any library names it may find in an object file. A high-level language compiler may add a library name to an object file to ensure that a default set of libraries is linked with the program. Using this option bypasses these default libraries and lets you name the libraries you want by including them on the LINK command line.

Format

/NODEFAULTLIBRARYSEARCH

The minimum abbreviation is **/NOD**.

Disabling Far Call Translations /NOFARCALLTRANSLATION

This option directs LINK to disable translation of intra-segment far calls. This option is only valid for the OS/2 mode.

Format

/NOFARCALLTRANSLATION

The minimum abbreviation is **/NOF**.

Remarks

This is the default (see FARCALLTRANSLATION).

The /NOGROUPASSOCIATION option causes the linker to process a certain class of fix-up routines in a manner compatible with previous versions of the linker. This option is provided primarily for compatibility with previous versions of other IBM language compilers.

This option is valid only for code linked to run in the DOS 3.3 environment or in the DOS environment of OS/2.

Format

/NOGROUPASSOCIATION

The minimum abbreviation is **/NOG**.

Remarks

The /NOGROUPASSOCIATION option should not be used with programs written in C.

Preserving Lowercase /NOIGNORECASE

The /NOIGNORECASE option directs LINK to treat uppercase and lowercase letters in symbol names as distinct letters. Normally, LINK considers uppercase and lowercase letters to be identical, treating the names *TWO*, *Two*, and *two* as the same. When you use the /NOIGNORECASE option, the linker treats *TWO*, *Two*, and *two* as different names.

Format

/NOIGNORECASE

The minimum abbreviation is **/NOI**.

Remarks

The /NOIGNORECASE option typically is used with object files created by high-level language compilers. Some compilers treat uppercase and lowercase letters as distinct letters and assume that the linker does the same.

Not Packing Code Segments /NOPACKCODE

This option directs LINK not to try to pack neighboring logical code segments into one physical segment.

This option is valid only for code linked to run in the OS/2 environment.

Format

/NOPACKCODE:

The minimum abbreviation is **/NOP**.

Remarks

/NOPACKCODE should be used to override the default, /PACKCODE.

For more information on packing, see the /PACKCODE option and "Rules for Segment Packing" in this appendix.

Setting the Overlay Interrupt /OVERLAYINTERRUPT

By default, the DOS interrupt number used for passing control to overlays is 3FH. The overlay interrupt option allows you to select a different interrupt number.

This option is valid only for code linked to run in the DOS 3.3 environment or in the DOS environment of OS/2.

Format

/OVERLAYINTERRUPT:*number*

The minimum abbreviation is **/O**.

Remarks

The *number* can be a decimal number from 0 to 255, an octal number from 0 to 0377, or a hexadecimal number from 0 to 0xFF. Numbers that conflict with DOS interrupts are not prohibited, but IBM does not recommend their use.

This option directs LINK to try to pack neighboring logical code segments into one physical segment.

This option is valid only for code linked to run in the OS/2 environment.

Format

/PACKCODE:[*packlimit*]

The minimum abbreviation is **/PAC**.

Remarks

The option /PACKCODE is the default. The option /NOPACKCODE should be used to override /PACKCODE.

The optional *packlimit* is the limit at which to stop packing. If no number is given, LINK uses 65530. For more information on packing, see "Rules for Segment Packing" in this appendix.

Pausing to Change Disks /PAUSE

The PAUSE option causes LINK to pause before writing the executable file to disk so that you can change disks.

Format

/PAUSE

The minimum abbreviation is **/PAU**.

Remarks

If you choose the /PAUSE option, the linker displays the following message before creating the run file:

About to generate .EXE file
Change diskette in drive *letter* **and press Enter**

The *letter* is the proper drive name. This message appears after the linker has read data from the object files and library files and after it has written data to the map file, if one was specified. LINK resumes processing when you press Enter. After LINK writes the executable file to disk, the following message appears:

Please replace original diskette
in drive *letter* **and press Enter**

Note: Do not remove the disk used for the temporary file, if one has been created. If the temporary disk message appears when you have specified the /PAUSE option, you should press Ctrl-Break to end the LINK session. Rearrange your files so that LINK can write the temporary file and the executable file to the same disk, then try again.

Setting the Maximum Number of Segments /SEGMENTS

The /SEGMENTS option directs the linker to process no more than *number* segments per program. If it meets more than the given limit, the linker displays an error message and stops linking. The /SEGMENTS option bypasses the default limit of 128 segments.

Format

/SEGMENTS:*number*

The minimum abbreviation is **/SE**.

Remarks

If you do not specify /SEGMENTS, the linker reserves enough storage space to process up to 128 segments. If your program has more than 128 segments, you must set the segment limit higher to increase the number of segments LINK can process. Set the segment limit lower if you get the following LINK error message:

Segment limit set too high

The *number* can be any integer value in the range 1 to 3072.

Example

This example sets the segment limit to 192:

```
LINK file/SE:192,,;
```

The next example sets the segment limit to 255 (X'FF'):

```
LINK moda+modb,run/SEGMENTS:0xff,ab,em+mlibfp;
```

Setting the Stack Size /STACK

The /STACK option sets the program stack to the number of bytes given by *size*. The linker automatically calculates the stack size of a program, basing the stack size on the size of any stack segments given in the object files. If you specify /STACK, the linker uses the given *size* in place of any value it may have calculated.

Format

/STACK:*size*

The minimum abbreviation is **/ST**.

Remarks

The *size* can be any positive integer value in the range 1 to 65535.

The stack size can also be changed with the STACKSIZE module definition file statement.

Example

The first example sets the stack size to 512 bytes.

```
LINK file/STACK:512,,;
```

The second example sets the stack size to 255 (X'FF') bytes.

```
LINK moda+modb,run/ST:0xFF,ab,\lib\start;
```

The final example sets the stack size to 24 (30 octal) bytes.

```
LINK startup+file/ST:030,,;
```

Warning of Incorrect Offset /WARNFIXUP

The /WARNFIXUP option directs the linker to issue a warning for each segment-relative fix up of location-type "offset," such that the segment is contained within a group but is not at the beginning of the group. The linker will include the displacement of the segment from the group in determining the final value of the fix up, opposite to what happens with DOS 3.3 executables.

Format

/WARNFIXUP

The minimum abbreviation is **/W**.

Advanced LINK Topics

Moving or Discarding Application Code Segments Under OS/2

OS/2 can move and discard application code segments, and move and swap data segments to take best advantage of storage.

Note: If the application must use a local heap, you can reserve heap space at run time by using the OS/2 service DosReallocSeg to increase the size of the automatic data segment. The *automatic data segment* is the group of logical segments, defined with the IBM Macro Assembler pseudo-op GROUP, named DGROUP.

Order of Segments

LINK copies segments to the executable file in the same order that it meets them in the object files. This order is maintained throughout the program unless the linker finds two or more segments having the same class name. Segments having identical class names belong to the same class type and are copied to the executable files as contiguous blocks.

Combined Segments

LINK uses combine types to tell if two or more segments that are sharing the same segment name should be one segment. The combine types are **public**, **stack**, **common**, and **private**.

- If a segment is combine type **public**, the linker combines it with any segments of the same name and class. When LINK combines segments, it makes the segments contiguous in storage; you can reach each address in the segments using an offset from one frame address. The result is the same as if the segments were defined as a whole in the source file.

  The linker preserves the align type of each segment in the combined segment. So, even though the individual segments compose a single, larger segment, the code and data in each segment retain the original align type of the segment. If LINK

tries to combine segments that total more than 64K bytes, it displays an error message.

- If a segment is combine type **stack**, the linker combines individual segments as it does for **public** combine types. A difference is that, for **stack** segments, LINK copies an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first **stack** segment (or combined **stack** segment) that LINK meets. If you use the **stack** type for **stack** segments, you need not give instructions to load the segment into the **SS** register.
- If a segment is combine type **common**, the linker combines it with any segments of the same name and class. When LINK combines **common** segments, it places the start of each segment at the same address. This creates a series of overlapping segments. The resulting combination segment has a length equal to the length of the longest individual segment.
- LINK assigns a default combine type **private** to any segments with no explicit combine type definition in the source file. LINK does not combine **private** segments.

Groups

A **group** lets LINK address non-contiguous segments of various classes relative to the same frame address. When LINK addresses **groups**, it adjusts all storage references to items in the **group** relative to one frame address.

Segments of a group need not be contiguous, belong to one class, or have the same combine type. All segments of the **group** must fit within 64K bytes of storage. For OS/2 mode executable objects, a group is synonymous with a OS/2 *selector* or a physical segment.

Groups do not affect the order of loading segments. You must use class names and enter object files in the correct order to guarantee contiguous segments. If the **group** is smaller than 64K bytes of storage, LINK may place segments that are not part of the **group** in the same storage area. LINK does not specifically check that all segments in a group fit within 64K of storage. If the segments are larger than the 64K byte maximum, the linker can produce a **fix-up overflow** error.

A description of **groups** and defining **groups** is in the *IBM Personal Computer Macro Assembler/2 Language Reference* book.

Fix ups

Once the linker knows the starting address of each segment in a program and establishes all segment combinations and groups, it can **fix up** any unresolved references to labels and variables. The linker computes an appropriate offset and segment address and replaces the temporary address values with the new values.

The size of the value that LINK computes depends on the type of reference. If LINK discovers an error in the anticipated size of the reference, it displays a fix-up overflow error message. This happens, for example, when a program tries to use a 16-bit offset to address an instruction in a segment with a different frame address. It also occurs when the segments in a group do not fit within a single, 64K block of storage.

LINK resolves four types of references:

- Short
- Near self-relative
- Near segment-relative
- Long.

- A **short** reference occurs in JMP instructions that try to pass control to labeled instructions that are in the same segment or group. The target instruction must be no longer than 128 bytes from the point of reference. The linker computes a signed, 8-bit number for this **short** reference. It displays an error message if the target instruction belongs to a different segment or group (different frame address). The linker also displays an error message if the distance from the frame address to the target is more than 128 bytes in either direction.
- A **near self-relative** reference occurs in instructions that get access to data relative to the same segment or group. The linker computes a 16-bit offset for this **near self-relative** reference. It displays an error message if the data resides in more than one segment or group.
- A **near segment-relative** reference occurs in instructions that attempt to get access to data either in a specified segment or group, or relative to a specified segment register. LINK com-

putes a 16-bit offset for this **near segment-relative** reference. It displays an error message if the offset of the target within the specified frame is greater than 64K bytes or less than 0 bytes. LINK also displays an error message if LINK cannot address the beginning of the canonical frame of the target.

- A **long** reference occurs in CALL instructions trying to access an instruction in another segment or group. LINK computes a 16-bit frame address and a 16-bit offset for this **long** reference. The linker displays an error message if the computed offset is greater than 64K or less than 0 bytes. The linker also displays an error message if LINK cannot address the beginning of the canonical frame of the target.

Rules for Segment Packing in LINK

When the linker produces an OS/2 executable object, it can pack distinct, adjacent segments into the same physical or file segment. Physical or file segments are represented by entries in the program segment table. The rules that LINK uses when packing segments follow:

- The limit of the total size of a set of segments packed into a file segment is 64K. LINK starts a new file segment while packing segments into a file when the size of the file segment reaches 64K.
- LINK packs adjacent segments only into a file segment.
- LINK packs segments in the same group into a file segment. LINK does not pack segments in different groups into a file segment. If a segment in one group occurs between segments in another group, an error occurs.
- /PACKCODE is the default. /NOPACKCODE should be used to override the default.
- If the segment type (CODE or DATA) is not the same in all the packed segments, LINK sets the file segment flags to NON-SHARED CODE.
- If LINK packs any segments that have an attribute of NON-SHARED, it sets the file segment flags NONSHARED CODE.
- If LINK packs any segments that are not ERONLY (execute/read only), it marks the entire file segment as not ERONLY.

- If LINK packs any segments that are PRELOAD, it designates the entire file segment as PRELOAD.

The Map File

The map file lists the names, load addresses, and lengths of all segments in a program. It also lists the names and load addresses of any groups in the program, the program's start address, and messages about any errors the linker encountered. If the /MAP LINK option is used, the map file lists the names and load addresses of all public symbols.

In the map file for programs that execute in the OS/2 environment, segment information has the general form:

```
PROGRAM_A

Start     Length          Name            Class

0001:0000 02C24H          _TEXT           CODE
0001:2C30 02502H          EMULATOR_TEXT   CODE
0001:5132 00000H          C_ETEXT         ENDCODE
0002:0000 00170H          EMULATOR_DATA   FAR_DATA
0003:0000 00036H          NULL            BEGDATA
0003:0036 00708H          _DATA           DATA
```

The Start column shows the address of the first byte in the segment, in the form segment number:offset. The segment numbers are indexes to the segment table of the executable file, and start from 1. The Length column shows the length of the segment in bytes. The Name column shows the name of the segment, and the Class column shows the class name of the segment.

Group information has the general form:

```
Origin          Group
0003:0          DGROUP
```

At the end of the listing file, the linker shows the address of the program entry point.

```
Program entry point at 0001:02A0
```

In the map file for programs that execute in the DOS environment, segment information has the general form:

```
Start  Stop   Length Name              Class

00000H 02B86H 02B87H _TEXT             CODE
02B90H 05091H 02502H EMULATOR_TEXT     CODE
05092H 05092H 00000H C_ETEXT           ENDCODE
050A0H 0520FH 00170H EMULATOR_DATA     FAR_DATA
05210H 05245H 00036H NULL              BEGDATA
05246H 05761H 0051CH _DATA             DATA
```

The Start column shows the address of the first byte in the segment. The number shown is the offset from the beginning of the program. This number is referred to as the frame number. The Stop column shows the address of the last byte in the segment. The Length column has the length of the segment in bytes. The Name column shows the name of the segment, and the Class column shows the class name of the segment.

Group information has the general form:

```
Origin        Group
0521:0        DGROUP
```

At the end of the listing file, the linker shows the address of the program entry point.

```
Program entry point at 0000:02A0
```

If you have specified the /MAP LINK option, the linker adds a public symbol list to the map file. Symbols are listed twice: once in alphabetical order, second, in load address order This list has the general form shown in the following example. The form is the same for programs that execute in either the OS/2 or DOS environments. For each symbol address, the number to the left of the colon (:) represents a segment number for the programs that execute in the OS/2 environment, and a frame number for programs that execute in the DOS environment.

```
Address                 Publics by Name

0003:071A       $i8_implicit_exp
0003:0718       $i8_inpbas
0001:287C       $i8_input
0003:0719       $i8_input_ws
0001:0CF6       $i8_output

Address                 Publics by Value

0000:0000  Imp  DOSWRITE        (DOSCALLS.138)
0000:0000  Imp  DOSDEVCONFIG    (DOSCALLS.52)
0000:0000  Imp  DOSEXIT         (DOSCALLS.5)
0001:0010       _main
0001:01B0       _countwords
0001:0238       _analyze
0001:02A0       _astart
0001:0368       _cintDIV
```

The first three symbols shown in the example under Publics by Value are imported public symbols and appear in map files created for programs that run only in the OS/2 environment.

Linker Error Messages and Limits

This section lists error messages produced by the IBM Linker.

Fatal errors cause the linker to stop running. Fatal error messages have the following format:

location: **error L1***xxx*: *message text*

Non-fatal errors indicate problems in the executable file. LINK produces the executable file (and sets the error bit in the header if for the OS/2 mode). Non-fatal error messages have the following format:

location: **error L2** *xxx*: *message text*

Warnings indicate possible problems in the executable file. LINK produces the executable file (it does not set the error-bit in the header for the OS/2 mode). Warnings have the following format:

location: **error L4***xxx*:
message text

In these messages, *location* is the input file associated with the error, or LINK if there is not input file. If the input file is a module definitions file, the line number will be included, as shown below:

foo.def(3): fatal error L1030:
missing internal name

If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ

The following error messages may appear when you link object files with LINK.

L1001 *option* **: option name ambiguous**
A unique option name does not appear after the option indicator (/). For example, the command

LINK /N main;

produces this error, because LINK cannot tell which of the three options beginning with the letter **N** is intended.

L1002 *option* **: unrecognized option name**
An unrecognized character followed the option indicator (/), as in the following example:

LINK /ABCDEF main;

L1003 *option* **: MAP symbol limit too high**
The specified symbol limit value following the MAP option is greater than 32767, or there is not enough memory to increase the limit to the requested value.

L1004 *option* **: invalid numeric value**
An incorrect value appeared for one of the linker options. For example, a character string is entered for an option that requires a numeric value.

L1005 *option* **: packing limit exceeds 65536 bytes**
The number following the /PACKCODE option is greater than 65536.

L1006 *option* **: stack size exceeds 65534 bytes**
The size you specified for the stack in the /STACK option of the LINK command is more than 65534 bytes.

L1007 *option* **: interrupt number exceeds 255**
You gave a number greater than 255 as a value for the /OVERLAYINTERRUPT option.

L1008 *option* **: segment limit set too high**
The specified limit on the /SEGMENTS option is greater than 3072 using the /SEGMENTS option.

L1009 *option* **: CPARMAXALLOC : illegal value**
The number you specified in the /CPARMAXALLOC option is not in the range 1 to 65535.

L1020 no object modules specified
You did not specify any object-file names to the linker.

L1021 cannot nest response files
A response file occurs within a response file.

L1022 response line too long
A line in a response file is longer than 127 characters.

L1023 terminated by user
You entered **Ctrl** + **C** or **Ctrl** + **Break**.

L1024 nested right parentheses
You typed the contents of an overlay incorrectly on the command line.

L1025 nested left parentheses
You typed the contents of an overlay incorrectly on the command line.

L1026 unmatched right parenthesis
A right parenthesis is missing from the contents specification of an overlay on the command line.

L1027 unmatched left parenthesis
A left parenthesis is missing from the contents specification of an overlay on the command line.

L1030 missing internal name
In the module definitions file, when specifying an import by entry number, you must give an internal name, so the linker can identify references to the import.

L1031 module description redefined
In the module definitions file, a module description specified with the DESCRIPTION keyword is given more than once.

L1032 module name redefined
In the module definitions file, the module name is defined more than once with the NAME or LIBRARY keyword.

L1040 too many exported entries
An attempt is made to export more than 3072 names.

L1041 resident-name table overflow
The total length of all resident names, plus three bytes per name, is greater than 65534.

L1042 nonresident-name table overflow
The total length of all nonresident names, plus three bytes per name, is greater than 65534.

L1043 relocation table overflow
There are more than 65536 load-time relocations for a single segment.

L1044 imported-name table overflow
The total length of all the imported names, plus one byte per name, is greater than 65534 bytes.

L1045 too many TYPDEF records
An object module contains more than 255 TYPDEF records. These records describe communal variables. This error can only appear with programs produced by compilers that support communal variables.

L1046 too many external symbols in one module
An object module specifies more than the limit of 1023 external symbols. Break the module into smaller parts.

L1047 too many group, segment, and class names in one module
The program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object files.

L1048 too many segments in one module
An object module has more than 255 segments. Split the module or combine segments.

L1049 too many segments
The program has more than the maximum number of segments. The SEGMENTS option specifies the maximum allowed number; the default is 128. Relink using the /SEGMENTS option with an appropriate number of segments.

L1050 too many groups in one module
The linker found more than 21 group definitions (GRPDEF) in a single module. Reduce the number of group definitions or split the module.

L1051 too many groups
The program defines more than 20 groups, not counting DGROUP. Reduce the number of groups.

L1052 too many libraries

An attempt is made to link with more than 32 libraries. Combine libraries, or use modules that require fewer libraries.

L1053 symbol table overflow

The program has more than 256K bytes of symbolic information, such as public, external, segment, group, class, and file names. Combine modules or segments and recreate the object files. Eliminate as many public symbols as possible.

L1054 requested segment limit too high

The linker does not have enough memory to allocate tables describing the number of segments requested (the default is 128 or the value specified with the /SEGMENTS option). Try linking again using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 too many overlays

The program defines more than 63 overlays.

L1057 data record too large

A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator (compiler or assembler) error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your authorized IBM Personal Computer dealer.

L1070 segment size exceeds 64K

A single segment contains more than 64K bytes of code or data. Try compiling, or assembling, and linking using the large model.

L1071 segment _TEXT larger than 65520 bytes

This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this is increased to 16 for alignment purposes.

L1072 common area longer than 65536 bytes

The program has more than 64K bytes of communal variables. This error cannot appear with object files produced by the IBM Macro Assembler. It occurs only with programs produced by IBM C or other compilers that support communal variables.

L1073 file-segment limit exceeded
There are more than 255 physical or file segments.

L1074 *name* : group larger than 64K bytes
A group contained segments which total more than 65536 bytes.

L1075 entry table larger than 65535 bytes
Because of an excessive number of entry names, you have exceeded a linker table size limit. Reduce the number of names in the modules you are linking.

L1080 cannot open list file
The disk or the root directory is full. Delete or move files to make space.

L1081 out of space for run file
The disk on which .EXE file is being written is full. Free more space on the disk and restart the linker.

L1082 stub .EXE file not found
The stub file specified in the module definitions file is not found.

L1083 cannot open run file
The disk or the root directory is full. Delete or move files to make space.

L1084 cannot create temporary file
The disk or root directory is full. Free more space in the directory and restart the linker.

L1085 cannot open temporary file
The disk or the root directory is full. Delete or move files to make space.

L1086 scratch file missing
Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1087 unexpected end-of-file on scratch file
The disk with the temporary linker-output file is removed.

L1088 out of space for list file
The disk on which the listing file is being written is full. Free more space on the disk and restart the linker.

L1089 *filename* **: cannot open response file**
The linker could not find the specified response file. This usually indicates a typing error.

L1090 cannot reopen list file
The original disk is not replaced at the prompt. Restart the linker.

L1091 unexpected end-of-file on library
The disk containing the library probably was removed. Replace the disk containing the library and run the linker again.

L1092 cannot open module definitions file
The specified module definitions file cannot be opened.

L1100 stub .EXE file invalid
The stub file specified in the definitions file is not a valid .EXE file.

L1101 invalid object module
One of the object modules is non-valid. If the error persists after recompiling, contact your authorized IBM Personal Computer dealer.

L1102 unexpected end-of-file
A non-valid format for a library was found.

L1103 attempt to access data outside segment bounds
A data record in an object module specified data extending beyond the end of a segment. This is a translator error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact your authorized IBM Personal Computer dealer.

L1104 *filename* **: not valid library**
The specified file is not a valid library file. This error causes the linker to stop running.

L1110 DosAllocHuge failed
Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1111 DosReallocHuge failed
Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1112 DosGetHugeShift failed
Internal error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1113 unresolved COMDEF; internal error
You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L1114 file not suitable for /EXEPACK; relink without
For the linked program, the size of the packed load image plus the packing overhead is larger than that of the unpacked load image. Relink without the EXEPACK option.

L2000 imported entry point
A MODEND, or starting address record, referred to an imported name. Imported program-starting addresses are not supported.

L2001 fix up(s) without data
A FIXUP record occurred without a data record immediately preceding it. This is probably a compiler error. Note the conditions and contact an authorized service coordinator.

L2002 fix up overflow near *number* **in frame seg** *segname* **target seg** *segname* **target offset** *number*
The following conditions can cause this error:

- A group is larger than 64K bytes.
- The program contains an intersegment short jump or intersegment short call.
- The name of a data item in the program conflicts with that of a subroutine in a library included in the link.
- An EXTRN declaration in an assembler-language source file appeared inside the body of a segment.

For example:

```
code    SEGMENT public  'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
        EXTRN   main:far
code    SEGMENT public  'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and recreate the object file.

L2003 Intersegment self-relative fix up
An intersegment self-relative fix up is not allowed.

L2004 LOBYTE-type fix up overflow
A LOBYTE fix up produced an address overflow.

L2005 fix up type unsupported
A fix up type occurred that is not supported by the linker. This is probably a compiler error. You should note the conditions when the error occurs and contact your authorized IBM Personal Computer dealer.

L2010 too many fix ups in LIDATA record
There are more fix ups applying to a LIDATA record than will fit in the linker's 1024-byte buffer. The buffer is divided between the data in the LIDATA record itself and run-time relocation items, which are 8 bytes apiece, so the maximum varies from 0 to 128. This is probably a compiler error.

L2011 *name* **: NEAR/HUGE conflict**
Conflicting NEAR and HUGE attributes are given for a communal variable. This error can occur only with programs produced by compilers that support communal variables.

L2012 *name* **: array-element size mismatch**
A far communal array is declared with two or more different array-element sizes (for example, an array declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the IBM Macro Assembler. It occurs only with IBM C and other compilers that support far communal arrays.

L2013 LIDATA record too large
A LIDATA record in an object module contains more than 512 bytes of data. Most likely, an assembly module contains a very complex structure definition or a series of deeply-nested DUP

operators. For example, the following structure definition causes this error:

```
alpha   DB     10DUP(11 DUP(12 DUP(13 DUP(...))))
```

Simplify the structure definition and reassemble. (LIDATA is a DOS term.)

L2020 no automatic data segment
No group named DGROUP is declared.

L2021 library instance data not supported in the DOS mode
The library module is directed to have instance data. This works in the OS/2 mode only.

L2022 *name* **alias** *internalname*: **export undefined**
A name is directed to be exported but is not defined anywhere.

L2023 *name* **alias** *internalname*: **export imported**
An imported name is directed to be exported.

L2024 *name* **: symbol already defined**
One of the special overlay symbols required for overlay support is defined by an object.

L2025 *name* **: symbol defined more than once**
Remove the extra symbol definition from the object file.

L2026 multiple definitions for entry ordinal *number*
More than one entry point name is assigned to the same ordinal.

L2027 *name* **: ordinal too large for export**
You tried to export more than 3072 names.

L2028 automatic data segment plus heap exceeds 64K
The size of DGROUP near data plus requested heap size is greater than 64K.

L2029 unresolved externals
One or more symbols are declared to be external in one or more modules, but they are not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message, as shown in the following example:

```
_exit in file(s)
 main.obj (main.c)
_fopen in files(s)
 fileio.obj(fileio.c) main.obj(main.c)
```

The name that comes before **in file(s)** is the unresolved external symbol. On the next line is a list of object modules which have

made references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 starting address not code (using class 'CODE')
You specified a starting address to the linker which is a segment that is not a CODE segment. Reclassify the segment to CODE, or correct the starting point.

L4001 frame-relative fix up, frame ignored
A fix up occurred with a frame segment different from the target segment where either the frame or the target segment is not absolute. Such a fix up is meaningless in the OS/2 mode, so the target segment is assumed for the frame segment.

L4002 frame-relative absolute fix up
A fix up occurred with a frame segment different from the target segment where both frame and target segments were absolute. This fix up is processed using base-offset arithmetic, but the warning is issued because the fix up may not be valid in OS/2 environment.

L4010 invalid alignment specification
The number following the **/ALIGNMENT** option is not a power of 2, or is not in numerical form.

L4011 PACKCODE value exceeding 65500 unreliable
Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4012 load-high disables EXEPACK
The options **/HIGH** and **/EXEPACK** are mutually exclusive.

L4013 invalid option for new-format executable file ignored
If an OS/2 environment program is being produced, the options **/CPARMAXALLOC**, **/DSALLOCATE**, **/EXEPACK**, **/NOGROUPASSOCIATION**, and **/OVERLAYINTERRUPT** are meaningless, and the linker ignores them.

L4014 invalid option for old-format executable file ignored
If an OS/2 format program is produced, the options **/ALIGNMENT**, **/NOFARCALLTRANSLATION**, and **/PACKCODE** are meaningless, and the linker ignores them.

L4020 *name* **: code-segment size exceeds 65500**
Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4021 no stack segment

The program does not contain a stack segment defined with STACK combine type. This message should not appear for modules compiled with the IBM C/2 Compiler but it could appear for an assembler-language module. Normally, every program should have a stack segment with the combine type specified as STACK. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type.

L4022 *name1, name2* **: groups overlap**

Two groups are defined such that one starts in the middle of another. This may occur if you defined segments in a module definitions file or assembly file and did not correctly order the segments by class.

L4023 *exportname* **: export internal-name conflict**

An exported name, or its associated internal name, conflict with an already-defined public symbol.

L4024 *name* **: multiple definitions for export name**

The name *name* is exported more than once with different internal names. All internal names except the first are ignored.

L4025 *name* **: import internal-name conflict**

An imported name, or its associated internal name, is also defined as an exported name. The import name is ignored. The conflict may come from a definition in either the module definition file or an object file.

L4026 *modulename* **: self-imported**

The module definitions file directed that a name be imported from the module being produced.

L4027 *name* **: multiple definitions for import internal-name**

An imported name, or its associated internal name, is imported more than once. The imported name is ignored after the first mention.

L4028 *name* **: segment already defined**

A segment is defined more than once with the same name in the module definitions file. Segments must have unique names for the linker. All definitions with the same name after the first are ignored.

L4029 *name* **: DGROUP segment converted to type data**
A segment which is a member of DGROUP is defined as type CODE in a module definition file or object file. This probably happened because a CLASS keyword in a SEGMENTS statement is not given.

L4030 *name* **: segment attributes changed to conform with automatic data segment**
The segment named *name* is defined in DGROUP, but the *shared* attribute is in conflict with the *instance* attribute. For example, the *shared* attribute is NONSHARED and the *instance* is SINGLE, or the *shared* attribute is SHARED and the *instance* attribute is MULTIPLE. The bad segment is forced to have the right *shared* attribute and the link continues. The image is not marked as having errors.

L4031 *name* **: segment declared in more than one group**
A segment is declared to be a member of two different groups. Correct the source file and recreate the object files.

L4032 *name* **: code-group size exceeds 65500 bytes**
Code segments of length 65501-65536 may be unreliable on the 80286 processor.

L4034 more than 239 overlay segments; extra put in root
You specified an overlay structure containing more than 239 segments. The extra segments have been assigned to the root overlay.

L4036 no automatic data segment
The program or dynalink library did not define a group named DGROUP, which is recognized by the linker as the automatic data segment.

L4040 NON-CONFIRMING : obsolete
In the module definitions file, NON-CONFORMING is a valid keyword for earlier versions of LINK and is now obsolete.

L4041 HUGE segments not yet supported
This feature is not implemented in the linker.

L4042 cannot open old version
An old version of the EXE file, specified with the OLD keyword in the module definitions file, could not be opened.

L4043 old version not segmented-executable format

The old version of the .EXE file, specified with the OLD keyword in the module definitions file, does not conform to segmented-executable format.

L4050 too many public symbols

The **/MAP** option is used to request a sorted listing of public symbols in the map file, but there were too many symbols to sort (the default is 2048 symbols). The linker produces an unsorted listing of the public symbols. Relink using **/MAP**:*number*.

L4051 *filename* **: cannot find library**

The linker could not find the specified file. Enter a new file name, a new path specification, or both.

L4053 VM.TMP : illegal file name; ignored

VM.TMP appears as an object-file name. Rename the file and rerun the linker.

L4054 *filename* **: cannot find file**

The linker could not find the specified file. Enter a new file name, a new path specification, or both.

Linker Limits

The table below summarizes the limits imposed by the linker. If you find one of these limits, adjust your program so the linker can accommodate it.

Item	Limit
Symbol table	256K
Load-time relocations (for OS/2 programs)	Default is 32K. If **/EXEPACK** is used, the maximum is 512K.
Public symbols	The range 7700-8700 can be used as a guideline for the maximum number of public symbols allowed; the actual maximum depends on the program.
External symbols per module	1023
Groups	Maximum number is 21, but the linker always defined DGROUP so the effective maximum is 20.
Overlays	63
Segments	128 by default; however, this maximum can be set as high as 3072 by using the **/SEGMENTS** option of the LINK command.
Libraries	32
Group definitions per module	21
Segments per module	255
Stack	64K

Module Definition File Statements

Notes:

1. Any line in the module definition file preceded by a semi-colon (;) is considered a comment. The linker ignores them.
2. The module definition file statement keywords must be in upper-case.

Statement	Description
CODE	Defines default attributes for code segments
DATA	Defines default attributes for data segments
DESCRIPTION	Inserts text into a program module
EXPORTS	Defines exported functions from dynamic link libraries
HEAPSIZE	Defines heap size in bytes
IMPORTS	Defines imported functions from dynamic link libraries
LIBRARY	Declares a dynamic link library
NAME	Declares a program module
OLD	Directs preservation of earlier ordinals
PROTMODE	Declares program to run in the OS/2 mode only
SEGMENTS	Defines attributes for code and data segments on a per segment basis
STACKSIZE	Defines stack size in bytes
STUB	Appends DOS executable file to the OS/2 program module

CODE
Defines the default attributes for code segments

Purpose
Defines a default attribute for code segments within a program module. The default attributes of individual segments can be overridden using the SEGMENTS statement.

Format
CODE *load executeonly iopl*

Remarks

load (optional) A keyword which specifies when a segment is to be loaded. Specify one of the following:

PRELOAD Load segment immediately.

LOADONCALL (default) Load segment on demand.

executeonly (optional) A keyword which specifies the access rights to code segments. Specify one of the following:

EXECUTEONLY Sets access rights to execute only.

EXECUTEREAD (default) Sets access rights to execute or read.

iopl (optional) The keyword *iopl* specifies whether or not code segments have I/O privilege.

IOPL Code segments have I/O privilege.

NOIOPL (default) Code segments do not have I/O privilege.

Example
```
CODE PRELOAD IOPL
```

DATA
Defines the default attributes for data segments

Purpose

Defines default attributes for data segments within a program module. All data segments are movable and swappable. With the exception of data segments specified with the option NONSHARED, the data segments in a dynamic link module are shared across all dynamic link modules in the library.

The default attributes of individual segments can be overridden using the SEGMENTS statement.

Format

DATA *load readonly instance iopl shared*

Remarks

load (optional) A keyword which specifies when a segment is to be loaded. Specify one of the following:

PRELOAD Load segment immediately.

LOADONCALL (default) Load segment on demand.

readonly (optional) A keyword which specifies the access rights to data segments.

READONLY Sets access rights to read only.

READWRITE (default) Sets access rights to read or write.

instance (optional) A keyword which defines how the automatic data segment, that is, the physical segment represented by DGROUP, is to be shared. Specify one of the following:

NONE Specifies that there is to be no automatic data segment.

SINGLE Specifies that the automatic data segment is to be shared by all instances of the module. This option is valid only for dynamic link libraries.

MULTIPLE Specifies that the automatic data segment is to be copied for each instance of the module.

Default: MULTIPLE for program modules and SINGLE for dynamic link libraries.

The linker ensures that the automatic data segment attribute specified by SINGLE or MULTIPLE matches the default sharing attribute of all data segments. By default, DATA SINGLE forces shared data. By default, DATA MULTIPLE forces nonshared data. Similarly, DATA SHARED forces DATA SINGLE; DATA NONSHARED forces DATA MULTIPLE.

If you specify contradictory statements such as DATA SINGLE and NONSHARED, the *instance* option will control the automatic data segment only. In this example, the automatic data segment is shared and all other data segments are nonshared.

iopl (optional) The keyword *iopl* specifies whether or not data segments have I/O privilege.

IOPL Data segments have I/O privilege.

NOIOPL (default) Data segments do not have I/O privilege.

shared (optional) Specifies whether or not a unique copy of the READWRITE data segments should be loaded for each process using a dynamic link library. You can specify the following values:

SHARED A single copy of each data segment is loaded. The copy is shared by all processes using the dynamic link library.

NONSHARED A unique copy of each READWRITE data segment is loaded for each process using the dynamic link library.

Default: NONSHARED for program modules and SHARED for dynamic link libraries.

Example

```
DATA SHARED READONLY PRELOAD
```

DESCRIPTION
Inserts text into a program module

Purpose

Inserts text, such as source control information or copyright information, into a program module.

Format

DESCRIPTION 'text'

Remarks

'text'	A one line string of ASCII characters enclosed in single quotation marks.

Example

```
DESCRIPTION 'Calendar, Version 1.00'
```

EXPORTS
Defines exported functions from dynamic link libraries

Purpose

Defines the names and attributes (functions) in the dynamic link libraries which are to be exported. The EXPORTS statement precedes the definitions. You can specify as many as 3072 export definitions, each on a separate line, after the EXPORTS statement. You must specify an export definition for every function in a dynamic link library.

Format

EXPORTS exportname=*internalname ordinal res iopl-parmwords*

Remarks

exportname — One or more ASCII characters which define the function name to be used by applications to access the exported function.

internalname — (optional) Defines the actual name of the function in the dynamic link library.

ordinal — (optional) An integer number which specifies the function's ordinal position within the dynamic link entry table. It is specified as *@ordinal*, where ordinal is the integer position number. If specified, the entrypoint can be called by either name or ordinal.

res — (optional) The keyword RESIDENTNAME specifies that the function's entry points should be kept resident in memory. Use this option only if you specified the ordinal option. If the call is by entry point rather than by ordinal, frequently used entry points keep resident in memory enables OS/2 to resolve calls more rapidly.

iopl-parmwords — (optional) A numeric value which you must specify for functions executing with I/O privilege. A function which executes with I/O privilege is allocated a 512-byte stack. When the function is called, the number of parameters (words) specified by *iopl-*

EXPORTS
Defines exported functions from dynamic link libraries

parmwords is copied from the caller's stack to the new stack.

Although the EXPORTS statement is normally used to identify functions in dynamic link libraries, the statement is also used for functions within program modules which execute with I/O privilege. When this is the case, the only valid arguments are exportname (to identify the function within the application) and *iopl-parmwords*.

Example

```
EXPORTS
        SampleRead
        StringIn=IntStringIn
        ScreenOut 6
        CharTest @3 RESIDENTNAME
```

HEAPSIZE
Defines heap size in bytes

Purpose
Defines the number of bytes an application needs for its local heap.

Format
HEAPSIZE *bytes*

Remarks

bytes	An (integer) number which specifies the local heap size in bytes.

Example
```
HEAPSIZE 8000
```

IMPORTS
Defines functions imported from dynamic link libraries

Purpose

Defines the names of functions to be imported from dynamic link libraries. External references to dynamic link libraries generally are resolved by specifying the .LIB files for the dynamic link library to the linker. The IMPORTS statement is an alternate method of resolving external references to dynamic link libraries. IMPORTS must precede the definitions. Specify each definition on a separate line.

Format

IMPORTS *internalname*=libraryname.entry

Remarks

internalname (optional) Specifies the name used by the application to call the function being imported. *internalname* is a unique name comprised of one or more ASCII characters.

libraryname The name of the dynamic link library which contains the function.

entry The name of the function to be imported. It can be either of the following:

.entryname The name of the function as it appears in the dynamic link library.

.entryordinal The ordinal value of the function. The ordinal value corresponds to the entry point in the dynamic link library.

You can import OS/2 functions in the DOSCALLS library only by using .entryordinal.

Note: .entryname references to DOSCALLS are not supported and will fail during the module load. Therefore, rather than directly importing OS/2 functions, link with DOSCALLS.LIB.

IMPORTS
Defines functions imported from dynamic link libraries

Example

```
IMPORTS
        Sample.SampleWrite
        Read=Sample.SampleRead
```

LIBRARY
Declares a dynamic link library

Purpose

Specifies that the executable file being created is a dynamic link library. It also specifies the type of library initialization required for the library.

Note: The LIBRARY statement tells LINK to build a dynamic link library. Do not associate the LIBRARY statement with .LIB files which are libraries of object modules.

Format

LIBRARY *libraryname initialization-type*

Remarks

libraryname (optional) Defines the name of the dynamic link library. After a dynamic link library has been loaded, *libraryname* is the name known to OS/2. The *libraryname* is not normally specified. The *libraryname* can be up to eight characters.

Use caution when specifying a *libraryname* that differs from the filename of the .DLL being created. Subsequent load requests, in the process of determining whether a .DLL has already been loaded, will recognize only this *libraryname* as the name of the loaded dynamic link library.

Default: The executable filename, not including the extension.

initialization-type (optional) Keyword specifying the type of library initialization required by the library module. This keyword is ignored if a library initialization routine is not defined for the library module. It must be one of the following:

- INITGLOBAL - called only once when the library module is loaded initially.
- INITINSTANCE - called once for each process which gains access to the library module.

Default: INITGLOBAL

When the dynamic link module is loaded, it (optionally) can call an initialization routine before calling the dynamic link library. See "Program Execution Control" on page 4-26 for additional information.

Specify either the NAME statement or the LIBRARY statement. You cannot specify both. If neither NAME nor LIBRARY is specified, the default is NAME. If you use the LIBRARY statement, it must be the first statement in the module definition file.

Example

```
LIBRARY MathCall
```

NAME
Declares a program module

Purpose
Specifies that the executable file being created is a program module.

Format
NAME *modulename*

Remarks

modulename (Optional) Defines the name of the program module. After a program module has been loaded, the name known to OS/2 is the *modulename*, which can have up to eight characters. The *modulename* is not normally specified.

Default: The executable filename, not including the extension.

Remarks
Specify either the NAME statement or the LIBRARY statement. You cannot specify both. If neither NAME nor LIBRARY is specified, the default is NAME. If you use the NAME statement, it must be the first statement in the module definition file.

Example

```
NAME Calendar
```

OLD
Specifies previous version of a dynamic link module

Purpose
Directs the linker to use export ordinals from the specified dynamic link library.

Format
OLD 'filename'

Remarks

If 'filename' is not found in the current directory, the linker looks in the list of directories in the user's PATH environment variable.

Exported names in this module matching exported names in the OLD module will be assigned ordinal values from the OLD module unless:

- The name in the OLD module does not have an assigned ordinal.
- An ordinal is explicitly assigned to the name in this dynamic link library.

The OLD statement is useful for preserving export ordinals across successive versions of a dynamic link module.

Example

```
OLD 'doscalls.dll'
```

PROTMODE
Sets a program module to run in the OS/2 environment

Purpose
Directs the linker to set the OS/2-bit in the OS/2 executable file header. PROTMODE specifies that the program module runs in the OS/2 environment only.

Format
PROTMODE

Remarks
PROTMODE designates that the executable file is not to be operated in the DOS mode; that is, it will not be "bound" using the BIND utility. If your program is never to run in the DOS mode, use PROTMODE to eliminate the floating-point fix ups and entry table entries for internal-reference fix ups from the executable file. Eliminating them removes wasted space in the file of OS/2-environment-only programs.

Example
```
PROTMODE
```

SEGMENTS
Defines the attributes of code and data segments

Purpose
Defines the attributes of code and data segments on a per segment basis. The specified parameters override the CODE and DATA statements' defaults. The SEGMENT statement must precede all other definitions, and you can specify any number of segment definitions after the SEGMENT statement. You must type each segment definition on a separate line.

Format
SEGMENTS segmentname *class load readonly executeonly iopl shared*

Remarks
segmentname Defines the code or data segment whose attributes are being specified. Enclose segmentname in single quotes if it is equal to a definition statement keyword such as

'DATA' or'CODE'.

class (optional) A keyword which specifies the class of the segment, such as

CLASS 'classname'.

If no 'classname' is given, the linker assumes class 'CODE'. Any segment whose classname ends in 'CODE' (case ignored) is recognized as a code segment by the linker: if a segment is defined without a CLASS option, it becomes type code.

load (optional) A keyword which specifies when a segment is to be loaded. Specify one of the following:

PRELOAD Load the segment immediately.

LOADONCALL (default) Load the segment on demand.

readonly (optional) A keyword which specifies the access rights to data segments.

SEGMENTS
Defines the attributes of code and data segments

READONLY Sets access rights to read only.

READWRITE (default) Sets access rights to read or write.

executeonly (optional) A keyword which specifies the access rights to code segments.

EXECUTEONLY Sets access rights to execute only.

EXECUTEREAD. (default) Sets access rights to execute or read.

iopl (optional) The keyword *iopl* specifies whether or not segments have I/O privilege. This option applies to both code and data segments.

IOPL Segments have I/O privilege.

NOIOPL (default) Segments do not have I/O privilege.

shared (optional) Specifies whether or not a unique copy of the READWRITE data segments should be loaded for each process using a dynamic link module. You can specify the following values:

SHARED A single copy of each data segment is loaded. The copy is shared by all processes using the dynamic link module.

NONSHARED A unique copy of each READWRITE data segment is loaded for each process using the dynamic link module.

Default: NONSHARED for program modules and SHARED for dynamic link libraries.

Example

```
SEGMENTS
   CSEG LOADONCALL
   CSEG2 EXECUTEREAD
   DSEG1 READONLY
   DSEG2 SHARED
```

Purpose

Defines the number of bytes an application needs for its stack. The value specified for STACKSIZE overrides the size of any stack segment defined in the application.

Format

STACKSIZE *bytes*

Remarks

bytes	An (integer) number which specifies the stack size.

Example

```
STACKSIZE 1024
```

STUB
Appends DOS executable file to the OS/2 program module

Purpose

Appends the DOS executable file to the top of the OS/2 program module being created. The stub is called if the OS/2 program module is executed under DOS. For example, the stub could display a warning message and terminate.

Format

STUB 'filename'

Remarks

'filename' The name of the DOS executable file.

If 'filename' is not found in the current directory, the linker looks in the list of directories in the user's PATH environment variable.

Example

```
STUB'astub.exe'
```

Glossary

abort. Cancel, end, fail, prematurely end, stop.

abnormal termination. Unusual cessation of processing prior to planned termination.

access. To obtain use of storage, a device, or a service.

access mode. A technique that is used to obtain a specific logical record from, or to place a specific logical record into, a file assigned to a mass storage device.

active session. Foreground session.

addressability. (1) The ability of a process to access code or data. (2) Size of an address space.

algorithm. (1) A finite set of well-defined rules for the solution of a problem in a finite number of steps. (2) Method of calculation.

allocate. To assign or use for a specific purpose. For example, to assign memory for a programs data segment.

alphanumeric character. Consisting of letters, numbers, and other characters, such as punctuation marks and mathematical symbols.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

American National Standards Institute. An organization sponsored by the Computer and Business Equipment Manufacturers Association for establishing voluntary industry standards.

ANSI. American National Standards Institute.

API. (1) Application Program Interface. (2) The OS/2 function calls.

application. A program or group of programs that apply to a particular business area, such as the Inventory Control or the Accounts Receivable application.

application program. A program used to perform an application or part of an application.

archive. (1) A copy of a data set that can be used if the original data is lost. (2) The storage of object files in a library.

ARF. Automatic Response File used in linking.

ArgPointer. The address of a set of argument strings.

argument. Numbers, letters, or words that change the execution of a command.

ASCII. American National Standard Code for Information Interchange, normally to refer to a string of characters.

ASCIIZ. American National Standard Code for Information Interchange, normally to refer to a string of ASCII characters where the string is terminated with a byte of 0.

ASM. Abbreviation for Assembler.

assembler. A computer program that translates an assembly language source code file into a form that can be executed.

asynchronous. (1) Without regular time relationship. (2) Unexpected or unpredictable with respect to the execution of a program's instructions.

attribute. A characteristic or property of one or more entities, for example, the attribute for a displayed field could be blinking.

automatic mode. A method of operation that, under specific conditions, does not require human intervention.

auxiliary. A process or device not under direct control of the processing unit.

bimodal. Pertaining to both the DOS mode and the OS/2 mode.

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1.
(2) Involving a choice of two conditions.

binary string. A sequence of consecutive binary digits.

bit. Either of the binary digits 0 or 1 used in computers to store information.

block. (1) To wait, usually for an I/O event to complete or for a resource to become available.
(2) A storage area used to hold information.

boot. Initial program load after reset.

BPB. BIOS Parameter Block

buffer. (1) A temporary storage unit, especially one that accepts information at one rate and delivers it at another rate. (2) An area of storage, temporarily reserved for performing input or output, from which data is read, or into which data is written.

buffer address. A numeric value denoting a unique memory location for a buffer.

byte. The amount of storage required to represent one character; a byte is 8 bits.

bytes per sector. The term used to identify the number of bytes that can be stored on a sector of a disk/diskette (i.e., 512).

CALL. The action of bringing a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and an entry point.

CF. Carry flag.

character. A letter, digit, or other symbol.

character display. A display that uses a character generator to display predefined character boxes of images (characters) on the screen. This kind of display cannot address the screen any less than one character box at a time.

character key. A keyboard key that allows the user to enter the character shown on the key.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character string. A sequence of consecutive characters.

character variable. The name of a character data item whose value may be assigned or changed while the program is running.

child. (1) Pertaining to a secured resource, either a file or library, that uses the parent resources. A child resource can have only one parent resource. (2) A process created by a parent process that shares resources of parent process.

child process. (1) A dependent process; contrast with parent process. (2) A process that is created by another process.

CLI. Assembly language instruction to disable processor interrupts. The effect is to prevent hardware interrupts from being recognized by the processor until an STI instruction is issued.

click. Press the mouse button (or press the mouse button twice in rapid succession).

client process. A process that uses some service or dynamic link library. A process is a client of the service or library.

close. (1) To end an activity and remove it from the display. (2) To release a device.

code. (1) Instructions for the computer. (2) To write instructions for the computer; to program. (3) A representation of a condition, such as an error code.

code page. The character encoding, for keyboard input and display and spooled printer output.

code point. A single character or shape defined in the code page and presented on the display or printer.

code segment. See *segment*

COM. (1) Represents one of the serial communications ports supported by OS/2 (COM1, COM2, COM3). (2) Communication.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

command processor. (1) A program executed to perform an operation specified by a command. (2) A system or user task that proc-

esses a set of commands from a queue.

compile. (1) To translate a program written in a high-level programming language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

condition. An expression in a program or procedure that can be evaluated to a value of either true or false when the program or procedure is running.

configuration. The group of machines, devices, and programs that make up a computer system.

constant. A data item with a value that does not change.

context. The environment in which a program executes.

contiguous. To be in actual contact with or touching along a boundary.

coprocessor. A microprocessor on an expansion card or the system board.

CS. Code Segment.

current. Active.

cursor. (1) A movable symbol (such as an underline) on a display, used to indicate to the user where the next typed character will be placed or where the next action will be directed. (2) A marker that indicates the current data access location within a file.

customize. To describe (to the system) the devices, programs, users, and user defaults for a particular data processing system.

cylinder. All fixed-disk or diskette tracks that can be read or written without moving the disk drive or diskette drive read/write mechanism.

DASD. Direct Access Storage Device.

data area. A storage area used by a program to hold information.

DB. Define byte. A MASM pseudo-op to declare a byte of memory.

DBCS. Double Byte Character Set

DD. Define Double Word. A MASM pseudo-op to declare a double word of memory.

dead key. Inactive key.

deallocate. To release a resource that is assigned to a specific task.

default value. A value stored in the system that is used when no other value is specified.

deinstall. To remove. Used as a device driver command.

delete. A function that enables data held in storage to be removed.

delimiter. (1) A flag that separates and organizes items of data. Synonymous with punctuation symbol, separator. (2) A string of one or more characters used to separate

or organize elements of computer programs or data, for example, parenthesis, blank character, arithmetic operator, if, "BEGIN". (3) A character that groups or separates words or values in a line of input.

deregister. To remove. Used as a device monitor command.

destination drive. The target drive in an operation involving two or more logical drives.

DevHlp. Refers to the OS/2 services available in writing device drivers.

device. An electrical or electronic machine that is designed for a specific purpose and that attaches to a computer, for example, a printer, plotter, or disk drive.

device attributes. Characteristics of a device, described in the device header of a device driver.

device BPB. Data structure used to describe DASD devices.

device driver. A program that operates a specific device, such as a printer, disk drive, or display.

device handle. Device identifier.

device name. A name reserved by the system or a device driver that refers to a specific device.

device type. The general name for a kind of device.

digit. Any of the numerals from 0 through 9.

directory. A type of file containing the names and controlling information for other files or other directories.

disable. (1) To make nonfunctional. (2) The state of a processing unit that prevents the occurrence of certain types of interruptions. (3) A state in which a transmission control unit or audio response unit can not accept incoming calls.

disk. A flat circular plate with a surface layer on which data can be stored by magnetic recording.

diskette. A thin, flexible magnetic plate that is permanently sealed in a protective cover. It can be used to store information copies from the disk or another diskette.

diskette drive. The mechanism used to read and write information on diskettes.

dispatch. To allocate time on a processor to jobs or tasks that are ready for execution.

display. (1) In word processing, a device for visual presentation of information on any temporary character-imaging device. (2) A visual presentation of data. (3) To present data visually.

display device. An output unit that gives a visual representation of data.

display screen. The part of the display device that displays information visually.

DMA. Direct Memory Access

DOS. (1) IBM Personal Computer Disk Operating System. (2) The DOS mode of OS/2.

double-byte character. A single symbol requiring two bytes to code; for example, Chinese characters.

dump. (1) To copy the contents of all or part of storage, usually to an output device. (2) Data that has been dumped.

dump diskette. A diskette that contains a dump or is prepared to receive a dump.

DW. Define Word. A MASM pseudo-op to declare a word of memory.

dynamic priority variation. The changes in a thread's priority based on the state of the system and the state of thread relative to the system.

EBCDIC. Extended Binary-Coded Decimal Interchange Code.

EBCDIC character. Any one of the symbols included in the 8-bit EBCDIC set.

enable. (1) To make functional. (2) The state of a processing unit that allows the occurrence of certain types of interruptions. (3) The state in which a transmission unit can accept incoming calls on a line.

environment. The settings for variables and paths set associated with each process.

execute/execution. Do, start, or run.

exit value. A numeric value that a command returns to indicate whether it completed successfully. Some commands return exit values that give other information, such as whether a file exists.

expression. A representation of a value. For example, variables and constants appearing alone or in combination with operators.

FAT. File Allocation Table.

FCB. (1) Function control block. (2) File control block. (3) Forms control buffer.

field. (1) An area in a record or panel used to contain a particular category of data. (2) The smallest component of a record that can be referred to by a name.

FIFO. (1) First-in-first-out. (2) A type of queue where the oldest elements in the queue are removed before any newer elements. (3) Order of transferred data packets.

file. A collection of related data that is stored and retrieved by an assigned name.

file handle. File identifier.

file pointer. An internal name that is a pointer to a structure containing information about a file.

file specification. The name and location of a file. A file specification consists of a drive specifier, a pathname, and a filename.

file system. The collection of files and file management structures on a physical or logical mass storage device.

filename. (1) The name used by a program to identify a file. (2) The portion of the identifying name that precedes the extension.

filespec. *See* File specification.

filter. A command that reads standard input data, modifies the data, and sends it to standard output.

first-in-first-out(FIFO). (1) A type of queue where the oldest elements in the queue are removed before any newer elements. (2) Order of transferred data packets.

fixed disk. A flat, circular, nonremovable plate with a surface layer on which data can be stored by magnetic recording.

flag. (1) A modifier that defines the action of a command. (2) The action or return from a command.

floppy disk. Deprecated term for diskette.

flush. Delete, erase, or remove.

foreground session. The session currently interacting with the user.

format. (1) A defined arrangement of such things as characters, fields, and lines, usually used for displays, printouts, or files. (2) The pattern which determines how data is recorded.

function. (1) A specific purpose of an entity, or its characteristic action. (2) A synonym for procedure.

graphic character. (1) A character, other than a control character, that is normally represented by a graphic. (2) A character that can be displayed or printed.

half-byte. Either the first or last four consecutive bits of a byte. A hexadecimal character represents a half-byte.

hard disk. Fixed disk.

hard error. (1) Error caused by the state of the hardware (e.g., printer off-line). (2) An unrecoverable error.

hertz (Hz). A unit of frequency equal to one cycle per second.

hexadecimal code. A code based on the radix 16, in which the digits 0 through 9 and the letters A through F represent the code.

hot key. A key recognized by the system as a request for a particular service or action. OS/2 has two hot keys: Ctrl + Esc to return to the Program Selector and Alt + Esc to switch between active sessions.

hung. Halted or stopped.

Hz. Abbreviation for hertz.

Init. Initialization, usually refers to an initialization routine.

initialize. To set counters, switches, addresses, or contents of storage to

zero or other starting values at the beginning of, or at prescribed points, in the operation of a computer routine.

input. (1) Pertaining to a device process, or channel involved in an input process, or to the data or states involved in an input process. (2) Data to be processed. (3) Synonymous with input data, input process.

input device. Physical devices used to provide data to a computer.

interactive. (1) Permitting continuous dialog between the user and the operating system. (2) Pertaining to an application in which each entry calls forth a response from a system or a program.

interactive processing. A processing method in which each system user action causes response from the program or the system.

interface. A shared boundary between two or more entities. An interface might be a hardware component to link two devices together or it might be a portion of storage or registers accessed by two or more computer programs.

interrupt-time. A generic term that refers to executing code as a result of an interrupt; the thread of execution does not belong to a process.

I/O. Input/output.

IPC. Interprocess Communications via semaphores, pipes, queues, etc.

IPL. (1) Initial program load; the first time the operating system is loaded into memory and initialized. (2) To reload and initialize the operating system via a power off/on or by pressing Ctrl+Alt+Del.

invoke. To activate a procedure at one of its entry points.

KB. Kilobyte.

kbd. keyboard.

KCB. (1) Keyboard Control Block. (2) A data area containing variables pertaining to a given logical keyboard, created by the KbdOpen API.

kill. To end or suppress a process.

Kilobyte. 1024 bytes.

label. (1) The name in the disk or diskette directory that identifies a file. (2) The field of an instruction that assigns a symbolic name to the location at which the instruction begins.

LIB. (1) An abbreviation for library. (2) Used as an extension on library files. (3) The Librarian utility which creates and manages Object Libraries.

library. A collection of functions, calls, subroutines, or other data.

linefeed. An ASCII character that causes an output device to move forward one line.

load. (1) To move data or programs into storage. (2) To place a diskette into a diskette drive. (3) To insert paper into a printer.

logical device. Redirected disk, file, printer, or other specific device.

memory. Storage on electronic chips. Examples of memory are random access memory, read only memory, or registers.

memory compaction. Relocating allocated storage segments into contiguous locations in order to place all free storage in one large block.

menu. A displayed list of items from which a user can make a selection.

message. A response from the system to inform the user of a condition which may affect further processing of a program.

module. A discrete programming unit that usually performs a specific task or set of tasks.

monitor. (1) A routine which examines input to a character device driver and can delete, modify, or expand the character before passing it back to the device driver. (2) A routine which filters the input/output stream.

multi. Two or more.

multiprogramming. (1) A mode of operation that provides for the interleaved execution of two or more computer programs by a single processor. (2) The processing of two or more programs at the same time.

multitasking. The concurrent execution of two or more tasks by a computer.

nest. To incorporate a structure or structures of some kind into a structure of the same kind. For example, to nest one loop (the nested loop) within another loop (the nesting loop); to nest one subroutine (the nested subroutine) within another subroutine (nesting subroutine.)

NPX. Numeric Coprocessor

NUL. Null character

null. Having no value, containing nothing.

null character (NUL). The character hex 00, used to represent the absence of a printed or displayed character.

numeric. Pertaining to any of the digits 0 through 9.

OBJ. A file name extension that identifies a relocatable object file.

object code. Machine-executable instruction, usually generated by a compiler from source code written in a higher level language. Consists of directly executable machine code. For programs that must be linked, object code consists of relocatable machine code.

object file. A program unit that is the output of an assembler or a compiler and is suitable for input to the linker.

object module. Synonym for object file.

octal. A base eight numbering system.

offset. The number of measuring units from an arbitrary starting point in a record, area, control block, or a segment to some other point.

online. (1) Pertaining to a user's ability to interact with a computer. (2) Pertaining to the operation of a functional unit that is under the continual control of a computer.

open. To make a file or device available to a program for processing.

operand. (1) An entity to which an operation is applied. (2) That which is operated upon. An operand is usually identified by the address part of an instruction. (3) Information entered with a command name to define the data on which a command processor operates and to control the execution of the command processor. (4) An expression to whose value an operator is applied.

operating system. Software that controls the running of programs; in addition, an operating system may provide services such a resource allocation, scheduling, input/output control, and data management.

option. A specification in a statement that can be used to influence the execution of the statement.

output. (1) Synonym for output data; output process. (2) The result of processing.

output device. The physical device used by a computer to present data to a user.

output file. A file that is opened by a program so that the program can write to that file.

parameter. Information supplied by the programmer or user to a command, or function. An attribute, value, or variable.

parent. The owning or creating process, as opposed to a child process.

parent directory. The directory one level above the current directory.

parm. Abbreviation for parameter.

PgmPointer. The address of an ASCIIZ string of the drive, directory path and filename of the program to be executed.

physical device. Device.

pipe. To direct the data so that the output from one process becomes the input to another process.

pixel. Picture element.

pop. Retrieving information from a stack.

preempt. To take control away from, such as, to interrupt the execution of a process to allow another process to execute.

print queue. A file containing a list of the names of files waiting to be printed.

print spooler. The program which manages the print queue.

priority. A rank assigned to a task that determines its precedence in receiving system resources.

procedure. (1) The course of action taken for the solution of a problem. (2) The description of the course of action taken for the solution of a problem.

process. A collection of system resources including one or more threads.

process ID. A unique number assigned to a process.

prompt. A displayed request for information or user action.

protect mode. A mode of 80286 memory addressing in which virtual addresses are mapped to physical addresses by on-chip circuitry.

pseudo. Artificial, simulated.

Pull. To remove from.

push. Placing information on a stack (to put on or in).

queue. A line or list formed by items waiting to be processed.

RAM semaphore. A simple, efficient form of semaphore used to serialize different threads of a single process.

random access. An access mode in which records can be read from, written to, or removed from a file in any order.

RAS. Reliability, Availability, and Serviceability.

redirection. The reassignment of the standard input and standard output devices.

refresh. The process of repeatedly producing a display image on a display space so that the image remains visible.

RET. Return.

return code. A code returned by a function, which the caller may use to influence the execution of succeeding instructions.

reverse video mode. A form of highlighting a character, field, or cursor by reversing the color of the character, field, or cursor with its background; for example, changing a red character on a black background to a black character on a red background.

routine. Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

run. To cause a program, utility, or other machine function to be performed.

run time. The elapsed time taken for the execution of a computer program. Synonymous with run duration.

sector. An area on a disk track or a diskette track reserved to record information

segment. A contiguous area of storage.

semaphore. A signal mechanism used to control access to system resources.

separator. A character used to separate parts of a command or file.

sequential access. An access method in which records are read from, written to, or removed from a file based on the physical order of the records.

session. A routing mechanism for user interaction via the console.

shared data segment. A data area created in memory that can be shared by programs.

shared memory. An OS/2 feature that allows system memory to be shared among processes.

source module. The source statements or codes that constitute the input to the compiler or assembler for translation.

source program. A set of instructions written in a programming language, that must be translated to machine language compiled before the program can be run.

spawn. Create, develop, generate.

stack. An area in storage that stores temporary register information, parameters, and return addresses of subroutines.

stack pointer. A register that contains the current location of the top of the stack.

standalone. Pertaining to operations that are independent of another device, program, or system.

standard input. The primary source of data going into a command. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can be a file or another command.

storage. (1) The location of saved information. (2) In contrast to memory, the saving of information on physical devices such as disk or tape.

string. A linear sequence of entities such as characters or physical elements. Examples of strings are alphabetic string, binary element string, bit string, character string, search string, and symbol string.

suballocation. The allocation of a part of one extent for occupancy by elements of a component different from the one occupying the remainder of the extent.

subdirectory. A directory contained within another directory in the file system hierarchy.

subroutine. (1) A sequence of statements that may be used in one or more computer programs and at one or more points in a computer

program. (2) A routine that can be part of another routine.

swap file. A file that contains segments of a program or data temporarily moved out of main storage.

synchronous. Pertaining to two or more processes that depend upon the occurrences of specific events such as common timing signals.

system. The computer and its associated devices and programs.

task-time. A generic term that refers to executing code as a thread within a process.

terminate. Conclude, end, finish, stop.

thread. A unit of execution within a process.

toggle. Change or switch.

transposition. To change the order as to reverse the order.

typeamatic key. A key that repeats its function when held down.

valid. Correct.

variable. A parameter with different values at any one time. The values are usually restricted to data type.

wildcard character. Global filename characters.

write protection. Restriction of writing into a data set, file, or storage area by a user or program not authorized to do so.

Index

Special Characters

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Numerics

International Business
Machines Corporation
P.O. Box 1328-W
Boca Raton
Florida 33429-1328

Printed in the
United States of America

84X1434