Microsoft®
Operating System/2

Update to Device Drivers Guide

Microsoft Corporation

Document No. 510830020-200-O00-1087
Part No. 01317

Update to Device Drivers Guide

This update provides the changes and corrections to the *Microsoft® Operating System/2 Device Drivers Guide* for this release of the MS® OS/2 Software Development Kit. The following list details the changes and corrections. Text for new or corrected material is provided as separate function or command pages, divided by chapter.

Changes to Chapter 3, "Device Driver Commands"

In Section 3.3, "MS OS/2 Device Driver Request Packets," a paragraph has been added.

Section 3.4, "MS OS/2 Device Driver Commands":

- **Nondestructive Read No Wait** (Command Code 5). A paragraph has been added.
- **Status** (Command Codes 6 and 10). Two paragraphs have been added.
- **Flush** (Command Codes 7 and 11). The Purpose section has been restated, and a paragraph has been added.
- **Generic IOCtl** (Command Code 16). A *Note* has been added.
- **Reset Media** (Command Code 17). The text has been reworded.
- **Port Access** (Command Code 21). This command has been removed.

Changes to Chapter 4, "Generic IOCtl Commands"

The following commands in Section 4.2, "Serial Device Control IOCtl Commands (Category 01H)," have been renumbered, as shown in Table 1.1:

Table 1.1

Renumbered Commands

Command Was	Command Should Be	Command Name
4AH	44H	Transmit Immediate
4CH	47H	Stop Transmit
4DH	48H	Start Transmit
6BH	64H	Return Communications Status

Section 4.4, "Keyboard Control IOCtl Commands (Category 04H)":

- **Functions 58H**, **5BH**, **5CH**, **78H**, and **79H** have been added.
- **Functions 50H**, **53H**, **56H**, **73H**, and **76H** have changed.

Section 4.5, "Printer Control IOCtl Commands (Category 05H)":

- **Functions 48H**, **69H**, and **6AH** have been added.

Changes to Chapter 5, "Device Helper Services"

Section 5.4, "Process Management":

- **DevDone**. The Purpose section has been restated, and a paragraph has been added.

Section 5.5, "Semaphore Management":

- **SemHandle**. Three paragraphs have been added.
- **SemRequest**. A paragraph has been added.

Section 5.6, "Request Queue Management":

- A paragraph has been added to the introduction.
- **AllocReqPacket**. A paragraph has been added.
- **FreeReqPacket**. A paragraph has been added.

Section 5.8, "Memory Management":

- **PhysToVirt**. New return values have been added, as well as a new paragraph.
- **VerifyAccess**. A parameter and its description have been added, and one parameter description has been changed.

Section 5.9, "Interrupt Handling":

- **EOI**. Two paragraphs have been added.
- **SetIRQ**. Two paragraphs have been added.
- **UnSetIRQ**. A sentence has been added.

Section 5.10, "Timer Services":

- **SetTimer**. Three sentences have been added.
- **Tick Count**. Two sentences have been added.

Section 5.11, "Monitor Management":

- **MonFlush**. A sentence and a paragraph have been added.
- **MonWrite**. Two sentences and a paragraph have been added.

Section 5.12, "System Services":

- **GetDosVar**. The table of DOS variables has been revised.
- **GrantPortAccess**. This function has been removed.
- **PortUsage**. This function has been removed.
- **SendEvent**. Two *Event* values and their meanings have been added to the *Event* list.

OS/2 Device-Driver Commands
(Chapter 3, Pages 75–108)

OS/2 Device-Driver Request Packets
(Section 3.3, Pages 80–83)

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

Microsoft Operating System/2 does not guarantee that the order of API requests issued by multiple threads will be preserved when the corresponding request packets arrive at the device driver. Multiple application threads or threads created by the **DosReadAsync** and **DosWriteAsync** functions can get blocked in the operating system. This allows a device-driver request packet for an API request by a subsequent unblocked thread to arrive out of order. A device driver is responsible for providing a synchronization mechanism between itself and application processes/threads if it supports multiple outstanding requests and if request-packet ordering must be preserved.

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

BYTE	Length of request block
BYTE	Block device unit code
BYTE	Command code
WORD	Status
4 BYTES	Reserved
DWORD	Queue linkage
n BYTES	Command-specific data

Figure 3.4 MS OS/2 Request Packet

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

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

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

The *Status* field describes the resultant state of the request. The *Status* field is detailed as in Figure 3.5:

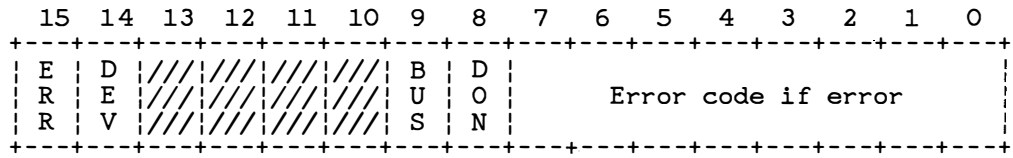


Figure 3.5 Request Packet Status Field

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

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

The following list details the error codes:

Code	Description
00H	Write-protect violation
01H	Unknown unit
02H	Device not ready
03H	Unknown command

04H	CRC error
05H	Bad drive request structure length
06H	Seek error
07H	Unknown media
08H	Sector not found
09H	Printer out of paper
0AH	Write fault
0BH	Read fault
0CH	General failure
0DH	Change disk (logical switch)
0EH	Reserved
0FH	Reserved
10H	Uncertain media
11H	Character I/O call interrupted
12H	Monitors not supported
13H	Invalid parameter

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

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

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

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

Remarks on Device Driver Errors

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

- When a drive-not-ready condition is detected

Note

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

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

MS OS/2 Device Driver Commands
(Section 3.4, Pages 84–108)

Nondestructive Read No Wait: Nondestructive Input (Command Code 5) *(Page 94)*

Purpose:

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

Format of Request Packet:

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

Remarks:

The device driver must perform the following actions:

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

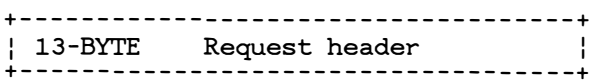
For input on a character device with a buffer: the busy bit is returned set to 0 to indicate that there is a copy of the first character in the buffer. The busy bit is returned set to 1 to indicate that no characters are in the buffer. This function allows the operating system to look ahead one input character without blocking in the device driver.

Status: Input or Output Status (Command Codes 6, 10) *(Page 95)*

Purpose:

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

Format of Request Packet:



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.

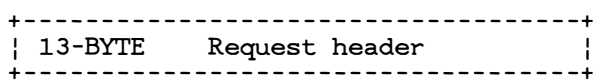
For input on a character device with a buffer: if the busy bit is returned set to 1, there are no characters currently in the device-driver buffer. If the busy bit is returned set to 0, there is at least one character in the device-driver buffer. In this case, a read of one character should not require the **Block** function. A device that does not have an input buffer in the device driver should always return the busy bit set to 0.

Flush: Input or Output Flush (Command Codes 7, 11) *(Page 96)*

Purpose:

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

Format of Request Packet:



Remarks:

The device driver must perform the following actions:

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

This call tells the device driver to flush (terminate) all pending requests that it has knowledge of. Its primary use is to flush the input queue on character devices.

Generic IOCtl: I/O Control for Devices (Command Code 16) *(Page 99)*

Purpose:

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

Format of Request Packet:

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

Remarks:

On entry, the request packet contains the following fields:

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

Note

Some IOCtl functions do not require data and/or parameters to be passed when the function is called. For these IOCtls, the parameter and data-buffer pointer fields of the request packet may each contain a DWORD of zero.

The device driver must perform the following actions:

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

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

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

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

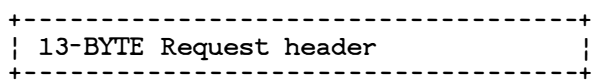
For more information about IOCtl commands, see Chapter 4, "Generic IOCtl Commands."

Reset Media: Reset Uncertain Media Condition (Command Code 17) *(Page 101)*

Purpose:

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

Format of Request Packet:



Remarks:

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

This command is called after the device driver returns an "uncertain media" error condition.

The device driver must perform the following action:

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

Generic IOCtl Commands
(Chapter 4, Pages 109–289)

Keyboard Control IOCtl Commands (Category 04H)
(Section 4.4, Pages 184–207)

Function 53H: Set Shift State
(Pages 188–189)

Purpose:

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

Parameter Packet Format:

```
+-----------------------------------+
|  WORD   Shift states              |
|-----------------------------------|
|  BYTE   NLS status                |
+-----------------------------------+
```

where:

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

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

10 Right CONTROL key down

11 Right ALT key down

12 SCROLL LOCK key down

13 NUMLOCK key down

14 CAPSLOCK key down

15 SYSREQ key down

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

Remarks:

Note

Many keyboards lack a SYSREQ key. It is therefore recommended that you not use bit 15 of the *Shift states* word.

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

Function 56H: Set Session Manager Hot Key
(Pages 192–193)

Purpose:

Function 56H changes the session manager hot key for which the keyboard device driver will scan.

Parameter Packet Format:

```
+-----------------------------------+
| WORD  Hot key                     |
|-----------------------------------|
| BYTE  Scan code make              |
|-----------------------------------|
| BYTE  Scan code break             |
|-----------------------------------|
| WORD  Hot key ID                  |
+-----------------------------------+
```

where:

Hot key is a Word specifying the setting for the session manager hot key. The bits in the *Hot key* word are defined as follows:

Bit	Meaning
0	Right SHIFT key down
1	Left SHIFT key down
2–7	Reserved = 0
8	Left CONTROL key down
9	Left ALT key down
10	Right CONTROL key down
11	Right ALT key down
12	SCROLL LOCK key down
13	NUMLOCK key down
14	CAPSLOCK key down
15	SYSREQ key down

Scan code make is a Byte containing the scan code of the hot key make.

Scan code break is a Byte containing the scan code of the hot key break.

Hot key ID is a Word set by the caller that identifies the session manager hot key.

Note

Scan code make and *Scan code break* are mutually exclusive; either may be specified, but not both. The use of either indicates when to recognize the hot key.

Remarks:

This request is used by the session manager to set a list of keyboard hot keys for which the keyboard device driver will scan. The new hot key applies to all screen groups. Up to sixteen hot keys can be defined by the session manager for use by the keyboard device driver.

Function 56H is successful only if it is performed by the process that initially called **Function 55H**, Set Foreground Screen Group.

The combination of the shift flags in the *Hot key* word and the scan-code bytes allow the session manager to set the hot key to a key combination such as ALT-ESC. The hot key is triggered on detection of the scan code for the hot-key break. Note that a hot key can be redefined by calling this function with the same *Hot key ID.*

Note

Many keyboards lack a SYSREQ key. It is therefore recommended that you not use bit 15 of the *Hot key* word.

Function 58H: Set KCB
(New)

Purpose:

Function 58H binds the specified logical keyboard (KCB) to the physical keyboard for this session.

Parameter Packet Format:

```
+-----------------------------------+
¦ DWORD  Handle                     ¦
+-----------------------------------+
```

where:

Handle is a double-word field identifying the logical keyboard's KCB.

Remarks:

None.

Function 5BH:
(New)

Purpose:

Function 5BH is reserved for Japanese keyboard support.

Function 5CH: Set NLS and Custom Code Page *(New)*

Purpose:

Function 5CH installs one of two possible code pages into the device driver. This IOCtl will update the number one or number two entry of the code-page control block. (Entry zero is the device-driver resident code page.)

Note

This IOCtl is similar to **Function 50H**, Set Translate Table, except it updates different entries in the code-function page control block. This IOCtl can be called from the real-mode (DOS 3.*x*) box.

Parameter Packet Format

DWORD	Selector:offset to code page
WORD	Code-page number
WORD	Number of table to Load

The number of the table to load is 1 or 2. If this number is –1, it indicates a custom code page, for which the segment containing the code page is locked. This option is not valid in real mode.

Remarks

None.

Function 73H: Get Shift State
(Pages 198–199)

Purpose:

Function 73H obtains the shift state of the screen group for the currently active process.

Data Packet Format:

```
+-----------------------------------+
¦ WORD  Shift states                ¦
+-----------------------------------+
```

where:

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

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

Remarks:

Note

Many keyboards lack a SYSREQ key. It is therefore recommended that you not use bit 15 of the *Shift states* word.

The *Shift state* is set by incoming keystrokes and by **Function 53H** calls.

Function 76H: Get Session Manager Hot Key
(Pages 204–206)

Purpose:

Function 76H returns the scan code that the keyboard device driver is using as the session manager hot key.

Parameter Packet Format:

```
+-----------------------------------+
| WORD  Hot key information         |
+-----------------------------------+
```

where:

Hot key information is a Word that specifies the type of information to return. On entry, this Word is defined as follows:

Value	Meaning
0	Return the maximum number of hot keys that the keyboard device driver can support.
1	Return the number of hot keys currently defined in the system and return the key information for each in the data packet.

Data Packet Format:

```
+-----------------------------------+
| WORD  Hot key                     |
|-----------------------------------|
| BYTE  Scan code make              |
|-----------------------------------|
| BYTE  Scan code break             |
|-----------------------------------|
| WORD  Hot key ID                  |
+-----------------------------------+
```

where:

Hot key is a word field containing the setting for the session manager hot key. The bits in the *Hot key* word are defined as follows:

Bit	Meaning
0	Right SHIFT key down
1	Left SHIFT key down
2–7	Reserved = 0
8	Left CONTROL key down
9	Left ALT key down
10	Right CONTROL key down
11	Right ALT key down
12	SCROLL LOCK key down
13	NUMLOCK key down
14	CAPSLOCK key down
15	SYSREQ key down

Scan code make is a Byte containing the scan code of the hot-key make.

Scan code break is a Byte containing the scan code of the hot-key break.

Note

Scan code make and *Scan code break* are mutually exclusive; either may be specified, but not both. The use of either indicates when the the hot-key is recognized.

Hot key ID is a Word set by the caller that identifies the session manager hot key.

Remarks:

Note

Many keyboards lack a SYSREQ key. It is therefore recommended that you not use bit 15 of the *Hot key* word.

If the Word in the parameter packet was 1 on entry, then one or more hot-key data structures (as defined in the data packet) are returned.

Function 76H first should be called with the *Hot key information* Word equal to zero to determine the maximum number of hot keys that can be supported by the device driver. The value returned should be used to determine the required size of the data buffer on a subsequent call (with *Hot key information* equal to one) to return the hot key data structures.

Function 78H: Get Code Page ID
(New)

Purpose:

Function 78H returns the code page being used by the current logical keyboard (KCB).

Data Packet Format:

WORD	Code page ID
WORD	Reserved

If the *Code page ID* field is zero, it indicates that PC US 437 is being used.

Remarks

None.

Function 79H: Translate Scan Code to ASCII *(New)*

Purpose:

Function 79H translates a scan code in a *CharData* record to an ASCII character.

Parameter Packet Format:

```
+--------------------------------------+
|       CharData Record                |
|--------------------------------------|
| WORD  KbdDDFlags                     |
|--------------------------------------|
| WORD  Xlate flags                    |
|--------------------------------------|
| WORD  Xlate shift state              |
|--------------------------------------|
| WORD  Reserved                       |
+--------------------------------------+
```

where:

The *Xlate flags* field contains the following information:

Bit	Meaning
0	Translation complete
1–7	Reserved = 0
8–15	Reserved = 0

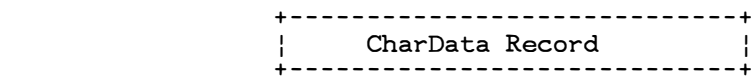
The *Xlate Shift State* field identifies the state of translation across successive calls. Initially, this Word should be set to zero. It should be reset to zero when the caller wants a new translation to start. It may take several calls to this IOCtl to complete the translation of a character, so this field should not be modified during a translation unless a "fresh start" to translation is desired. The field is set to zero at the completion of translation.

On entry, the parameter list contains the following fields:

Code page ID is a Word that specifies the code page to use for translation. If this field is zero, use the table specified by the active keyboard-control block.

The second field is a reserved Word and must be set to zero.

Data Packet Format



On return, the *CharData* record contains the ASCII field, translated based on the contents of the scan-code field.

Remarks:

You may specify a code page to use for translation. Otherwise, the code page of the active KCB will be used.

Printer Control IOCtl Commands (Category 05H) *(Section 4.5, Pages 208–214)*

Function 48H: Activate Font *(New)*

Purpose:

Function 48H activates a font for printing.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE   Command information       |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| WORD  Code Page                   |
+-----------------------------------+
| WORD  Font ID                     |
+-----------------------------------+
```

where:

Code page is a Word that specifies the value of the code page to make active:

Value	Meaning
0000H	If both the *Code page* and *Font ID* values are zero, set the printer to the hardware-default code page and font.
0001H–FFFFH	Valid code-page numbers

Font ID is a Word that specifies the ID number of the font to make active:

Value	Meaning
0000H	If both the *Code page* and *Font ID* values are zero, set the printer to the hardware-default code page and font.
	If only the *Font ID* value is zero, any font within the specified code page is acceptable.
0001H–FFFFH	Valid font ID numbers; font types defined by the font-file definitions.

Returns:

If **AX** = 0, no error.

Else, **AX** = Error code:

- Code-page and font switching are not active.
- Unable to access the specified font file.
- Unable to locate or activate the specified code page.
- Unable to locate or activate the specified font.

Function 69H: Query Active Font
(New)

Purpose:

Function 69H determines which code page and font are currently active.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE   Command information       |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| WORD  Code page                   |
+-----------------------------------+
| WORD  Font ID                     |
+-----------------------------------+
```

where:

Code page is a Word that, on return, specifies the number of the currently-active code page:

Value	Meaning
0000H	If both the *Code page* and *Font ID* values are zero, set the printer to the hardware-default code page and font.
0001H–FFFFH	Valid code-page numbers.

Font ID is a Word that, on return, specifies the ID number of the currently active font.

Value	Meaning
0000H	If both the *Code page* and *Font ID* values are zero, set the printer to the hardware-default code page and font. If only the *Font ID* is specified to be zero, any font within the specified code page is acceptable.

0001H–FFFFH	Valid font ID numbers; font types defined by the font-file definitions.

Returns:

If **AX** = 0, no error.

Else, **AX** = Error code:

- Code-page and font switching are not active.

Function 6AH: Verify Font
(New)

Purpose:

Function 6AH verifies that a particular code page and font are available for the specified printer.

Parameter Packet Format:

```
+-----------------------------------+
|  BYTE   Command information       |
+-----------------------------------+
```

where:

Command information is a reserved Byte that must be zero.

Data Packet Format:

```
+-----------------------------------+
| WORD  Code page                   |
+-----------------------------------+
| WORD  Font ID                     |
+-----------------------------------+
```

Remarks:

Code page is a Word that specifies the *Code page* number to verify. Values range from 0 through 65,535.

Font ID is a Word that specifies the *Font ID* value to verify. Values range from 0 through 65,535.

Note

A value of zero for both the *Code page* and the *Font ID* values indicates the hardware-default code page and font. This combination is always valid.

Returns:

If **AX** = 0, no error. *Code page* is valid.

Else, **AX** = Error code:

- Code-page and font switching are not active.
- *Code page* is not valid.
- *Font ID* is not valid.

Device Helper Services
(Chapter 5, Pages 291–391)

Process Management
(Section 5.4, Pages 303–311)

DevDone: Flag I/O Complete
(Page 307)

Purpose:

The **DevDone** function is called by a device driver at interrupt time to signify that a request that was returned incomplete to the kernel at strategy time has been completed.

Calling Sequence:

```
LES  BX,RequestPacket          ;Pointer to I/O request packet.
MOV  DL,DEVHLP_DEVDONE
CALL [Device_Help]
```

where:

RequestPacket contains a pointer to an I/O request packet.

Returns:

None.

Remarks:

Since the virtual address of a request packet is valid in both real and protected mode, the device driver may pass the request packet pointer to **DevDone** without being sensitive to the mode.

DevDone is typically called from the device interrupt routine. The device driver should set any error flags in the *Status* field of the request packet before calling the routine.

DevDone does not apply to request packets allocated by the **AllocReqPacket** function call.

The device driver does not have to call **DevDone** for requests at strategy time. Such requests should return with the done bit set in their request packet.

See Also:

AllocReqPacket

Semaphore Management
(Section 5.5, Pages 312–319)

SemHandle: Get Semaphore Handle
(Pages 315–317)

Purpose:

The **SemHandle** function provides a semaphore handle to the device driver.

Calling Sequence:

```
MOV  BX,SemHandleLow      ;Semaphore identifier
MOV  AX,SemHandleHigh     ;
MOV  DH,UsageFlag         ;Indicates if in use
MOV  DL,DEVHLP_SEMHANDLE
CALL [Device_Help]
```

where:

SemHandleLow and *SemHandleHigh* specify the low and high words of the semaphore handle.

UsageFlag is a flag that indicates whether or not the requested semaphore is in use. *UsageFlag* has one of the following values:

Value	Meaning
0	Not-in-use
1	In-use

Returns:

C —cleared if no error.

AX:BX is set to the system handle for the semaphore.

C —set if error.

AX = Error code:

- Invalid semaphore handle.

Remarks:

This function converts the semaphore handle (or user *key*), provided by the caller of the device driver, to a system handle that the device driver may use. This handle then becomes the key that the device driver uses to reference the system semaphore, allowing the system semaphore to be referenced at interrupt time by the device driver. The device driver also uses this key when it is finished with the system semaphore: To indicate that it is finished with the system semaphore, the device driver must call **SemHandle** with the *UsageFlag* which indicates when the device driver is finished.

SemHandle is called at task time to indicate that the system semaphore is "in-use," and is called at either task time or interrupt time to indicate that the system semaphore is "not-in-use." "In-use" means that the device driver may be referencing the system semaphore. "Not-in-use" means that the device driver has finished using the system semaphore and will not be referencing it again.

The "key" of a RAM semaphore is its *virtual address*, where virtual address is the generic term for both real- and protected-mode address forms (segment:offset, selector:offset). **SemHandle** may be used for RAM semaphores, but since RAM semaphores have no system handles, **SemHandle** will simply return the RAM semaphore key back to the caller.

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 from the one passed to the function, the device driver can determine that it is a handle for a system semaphore.

If this function returns a carry flag, the device driver should issue the **DevHlp_VerifyAccess** request with read/write access set in the *TypeOfAccess* parameter before determining that this semaphore 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 set a system semaphore "in-use" because:

- The caller-supplied semaphore handle refers to task-specific system semaphore structures. These structures are not available at interrupt time, so **SemHandle** converts the task-specific handle to a system-specific handle. For uniformity, the other semaphore **DevHlp** functions accept only system-specific handles, regardless of the mode.

- An application could delete a system semaphore while the device driver is using it. If a second application were to create a system semaphore soon after, the system structure that had been used by the original semaphore could be reassigned. A device driver that then tried to manipulate the original process's semaphore would inadvertently manipulate the new process's semaphore. For this reason, the **SemHandle** "in-use" indicator increments a counter so that even though the calling thread may still delete its task-specific reference to the semaphore, the semaphore still remains in the system structures.

A device driver must subsequently call **SemHandle** with "not-in-use" when it has finished using the semaphore, so that the system semaphore structure can be freed. For every call indicating "not-in-use," there must be a matching call indicating "in-use."

When this function is called, it can change the state of the interrupt flag. You should not depend on this state remaining unchanged.

SemRequest: Claim Semaphore
(Pages 318–319)

Purpose:

The **SemRequest** function claims a semaphore.

Calling Sequence:

```
MOV  BX,SemHandleLow      ;Semaphore handle
MOV  AX,SemHandleHigh     ;
MOV  CX,SemTimeoutLow     ;Time-out value
MOV  DI,SemTimeoutHigh    ;in milliseconds
MOV  DL,DEVHLP_SEMREQUEST
CALL [Device_Help]
```

where:

SemHandleLow and *SemHandleHigh* are the low and high words of the semaphore handle being claimed.

SemTimeoutLow and *SemTimeoutHigh* specify the values for the time-out limit which may be one of the following:

Value	Meaning
–1	Wait forever, time-out never occurs
0	No Wait if semaphore is owned
0	Time-out limit

Returns:

C —cleared if no error, set if error.

AX = Error code:

- Semaphore time-out.
- Interrupt.
- Semaphore owner died.
- Invalid handle.
- Too many semaphore requests.

Remarks:

SemRequest checks the status of a semaphore. If it is unowned, then **SemRequest** sets it as owned and returns immediately to the caller. If the semaphore is owned, **SemRequest** will optionally block the device driver thread until the semaphore is released or until a time-out occurs, then try again. The time-out parameter places an upper bound on the amount of time for **SemRequest** to block before returning to the requesting device driver thread.

The semaphore handle for a RAM semaphore is the *virtual address* of the double-word of storage allocated for the semaphore. Virtual address is a generic term used for addresses: segment:offset for real mode, selector:offset for protected mode.

If the device driver references the RAM semaphore at interrupt time, it must manage the addressability to the RAM semaphore. But for a system semaphore, the handle must be passed to the device driver by the caller via a Generic IOCtl call. By using the **SemHandle** function, the device driver must then convert the caller's handle to a system handle.

Note

The **SemRequest** function is valid in user mode only for RAM semaphores. System semaphores are not available for use in user mode by device drivers.

When this function is called, it can change the state of the interrupt flag. You should not depend on this state remaining unchanged.

See Also:

SemHandle

Request Queue Management
(Section 5.6, Pages 320–329)

The functions described in this section are listed as follows, along with a brief description of each:

Function	Description
AllocReqPacket	Allocate Request Packet
FreeReqPacket	Free Allocated Request Packet
PullParticular	Pull Specific Request Packet from Queue
PullReqPacket	Pull Request Packet from Queue
PushReqPacket	Push Request Packet onto Queue
SortReqPacket	Insert Request in Sorted Order

These functions provide simple linked-list management that allows device drivers to easily maintain a list or queue of request packets to be serviced. Prior to using these functions, the device driver must allocate and initialize a Dword queue header to zero.

Typical use of these functions is for the device driver to queue request packets that cannot be serviced immediately due to the device being busy. For example, a disk device driver that supports more than one device would maintain a request packet chain for each device.

Since the pointer to a request packet is bimodal, the device driver can use its own request-queue management to place and maintain the request-packet pointers in the linkage fields. The queue header points to the next packet to be pulled off the queue, and a queue header value of zero indicates an empty request queue. The *Queue linkage* field of the request packet is used to contain the linked list pointer to the next packet in the queue. A *Queue linkage* field containing zero indicates the end of the list.

AllocReqPacket: Allocate Request Packet
(Pages 321–322)

Purpose:

The **AllocReqPacket** function returns a pointer to a request packet.

Calling Sequence:

```
MOV DH,WaitFlag              ;Wait for available request packet
MOV DL,DEVHLP_ALLOCREQPACKET
CALL [Device_Help]
```

where:

WaitFlag is a flag that specifies whether to wait for an available request packet. *WaitFlag* has one of the following values:

Value	Meaning
0	Wait for request packet
1	No Wait, return immediately

Returns:

C—cleared if a request packet was allocated.

ES:BX is set to the address of the allocated request packet.

C—set if a request packet was not allocated.

Remarks:

AllocReqPacket returns a bimodal pointer to a maximal-sized request packet. The bimodal pointer is a virtual address that is valid for both real and protected modes.

Some device drivers, notably the disk device driver, need to have additional request packets to service task-time requests. Since device drivers are bimodal, they cannot use a packet residing in their data segment because the resulting pointer is not bimodal.

Request packets that were allocated by an **AllocReqPacket** can be placed in the request packet queue. Request packets allocated in this manner should be returned to the kernel as soon as possible via the

FreeReqPacket function. As a whole, the system has a limited number of request packets, so it is important that a device driver not allocate request packets to hold them for future use.

When this function is called, it can change the state of the interrupt flag. You should not depend on this state remaining unchanged.

See Also:

FreeReqPacket

FreeReqPacket: Free Allocated Request Packet
(Page 323)

Purpose:

The **FreeReqPacket** function releases a request packet previously allocated by the **AllocReqPacket** function.

Calling Sequence:

```
LES  BX,RequestPacket          ;Pointer to previous request packet
MOV  DL,DEVHLP_FREEREQPACKET
CALL [Device_Help]
```

where:

RequestPacket is a pointer to the request packet that was requested previously.

Returns:

None.

Remarks:

The device driver should free a request packet only if it has been allocated previously by the **AllocReqPacket** function. The **DevDone** function should not be used to return an allocated request packet.

The system has a limited number of request packets, so it is important that a device driver not allocate request packets to hold them for future use.

When this function is called, it can change the state of the interrupt flag. You should not depend on this state remaining unchanged.

See Also:

AllocReqPacket, DevDone

Memory Management
(Section 5.8, Pages 336–355)

PhysToVirt: Map Physical Address to Virtual Address
(Pages 345–349)

Purpose:

In real mode, the **PhysToVirt** function converts a 32-bit address to a segment:offset pair. In protected mode, **PhysToVirt** converts a 32-bit address to a valid selector:offset pair.

Calling Sequence:

```
MOV  BX,AddressLow          ;32-bit physical address
MOV  AX,AddressHigh         ;
MOV  CX,Length              ;Length of segment
MOV  DH,Result              ;Leave result
MOV  DL,DEVHLP_PHYSTOVIRT
CALL [Device_Help]
```

where:

AddressLow and *AddressHigh* are the low and high words of the 32-bit physical address to be converted.

Length should be set to the length, in bytes, of the segment transfer.

Result specifies where the virtual address result of the function will be returned. It is one of the following:

Value	Meaning
0	Return result in DS:SI
1	Return result in ES:DI

Returns:

C—set if error.

AX = Error code:

- Invalid address.

C—cleared if successful.

DH set to 0 on input:

- **DS:SI**

  Valid virtual address.

- **ES**

  If no mode switch, **ES** is preserved.

  Mode switch: if **ES** contains the address of the device-driver data segment on input, it is converted to a valid virtual address. Otherwise, it is set to zero.

DH set to 1 on input:

- **ES:DI**

  Valid virtual address.

- **DS**

  If no mode switch, **DS** is preserved.

  Mode switch: if **DS** contains the address of the device-driver data segment on input, it is 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.

Remarks:

Note

DS must point to the device driver's header when calling **PhysToVirt** the first time or when calling **PhysToVirt** the first time after calling **UnPhysToVirt**.

PhysToVirt provides addressability to data for bimodal operations performed during task time and interrupt time. The interrupt handler of a bimodal device driver must be able to address data buffers regardless of the context of the current process. This is true because the current LDT

(Local Descriptor Table) will not necessarily address the data space that contains the data buffer that the interrupt handler needs to access.

This function performs mode-dependent addressing on behalf of a device driver, relieving it of the need to recognize the CPU mode and the subsequent effects on accessing memory. However, this function is essential when the device driver needs to access a memory location at both task and interrupt time. This restriction applies because the context at interrupt time may differ from that at task time.

UnPhysToVirt is not required whenever **PhysToVirt** is used except as follows:

- When use of the converted address is ended (no more **PhysToVirt** calls)
- Before the procedure that issued **PhysToVirt** returns to its caller

In addition, multiple **PhysToVirt** calls may be performed prior to issuing the **UnPhysToVirt** call. Only one call to **UnPhysToVirt** is needed.

PhysToVirt leaves its *result* in either **ES:DI** or **DS:SI**, giving the caller the ability to move strings in either direction. Upon return, interrupts are off if the processor is in real mode and if the physical address is above one megabyte.

Typical use of **PhysToVirt** function is to convert the physical address of a buffer to a virtual address so that data may be transferred in or out at interrupt time.

PhysToVirt is guaranteed to preserve registers **CS**, **SS**, **SP**, and **DS** if called with the **DH** register equal to 1, or **ES** if called with the **DH** register equal to zero.

The only exception to this guarantee is when a mode switch occurs. If the system decides to switch to protected mode for an address that lies above the one-megabyte address boundary, and if the current mode is real mode, then:

- The segment addresses in the **CS** and **SS** registers are set for the current mode.
- The **SP** register is reserved.
- The **DS** register is set for the current mode only if it contains the data-segment value of the device driver and is not being used for the converted address.

Note

In the event of a mode switch, any previously stored address pointers that contain the **DS** register for the device driver data segment must be stored again by the device driver. The zero flag (**ZF**) is set of a change in address mode occurred. In this case, the device driver must recalculate and store again any buffer addresses that were previously saved.

If a **PhysToVirt** call had previously been done with the address mode unchanged, and if a subsequent **PhysToVirt** requires a switch to protected mode, then the previously converted **PhysToVirt** address is considered invalid for the current mode; **PhysToVirt** must be reissued to recalculate the address.

When **PhysToVirt** is being used to recalculate an address after a mode switch occurs, it (the second **PhysToVirt**), will not cause a mode switch. The previous address is then valid and preserved (as long as the recalculation uses the opposite segment register from the one that originally caused the mode switch).

The pool of temporary selectors used by **PhysToVirt** in protected mode is not dynamically extendable. The converted addresses are valid as long as the device driver does not relinquish control via **Block**, **Yield**, or **RET**. An interrupt handler may use converted addresses prior to its EOI, with interrupts enabled. If an interrupt handler needs to use converted addresses after its EOI, it must protect the converted addresses by running with interrupts disabled.

The segment length parameter may be set to the length of the transfer.

Hint

For performance reasons, a device driver should try to optimize its usage of **PhysToVirt** and **UnPhysToVirt**. For the first **PhysToVirt** call that the device driver makes, it should pick the address that is likely to cause a mode switch and use the **ES** register. This would permit the mode switch to take place and retain the driver's data segment in the **DS** register.

The device driver must not enable interrupts or change the returned segment register (**ES** or **DS**) before it has finished using the returned value. The *value* returned in the segment register has no physical meaning, so the caller of the **PhysToVirt** function should not have reason to examine it. While the pointer(s) generated by **PhysToVirt** are in use, the device driver may call only for another **PhysToVirt**. It may not call any other **DevHlp** routines, since they may not preserve the special **ES/DS** values.

On completing the operation with the virtual address, the device driver must restore the previous interrupt state. The returned virtual address will not be valid after a **Block**.

See Also:

Block, UnPhysToVirt, Yield

VerifyAccess: Verify Memory Access
(Pages 353–354)

Purpose

The **VerifyAccess** function verifies whether the user process has the correct access rights for the memory that it passed to the device driver. If the process does not have the correct access rights, it will be terminated.

Calling Sequence:

```
MOV AX,Segment              ;Selector or segment
MOV CX,MemLength            ;Length of memory area
MOV DI,MemOffset            ;Offset to memory area
MOV DH,TypeOfAccess         ;Verify read-only or read/write
MOV DL,DEVHLP_VERIFYACCESS
CALL [Device_Help]
```

where:

Segment is the selector or segment of the memory segment to be verified for access.

MemLength is the length, in bytes, of the memory area to be verified for access. Zero means 64KB.

MemOffset is the offset to the memory segment.

TypeOfAccess specifies access privileges to the memory segment. It has one of the following values:

Value	Meaning
0	Read access
1	Read/write access

Returns:

C —cleared if no error, access verified; set if error, access attempt failed.

Remarks:

A device driver can receive an address to memory as part of a Generic IOCtl request from a process. Since the operating system cannot verify addresses embedded in the IOCtl call, the device driver must request verification to prevent itself from accidentally destroying memory for a user process. If the verification test fails, **VerifyAccess** terminates the process.

Note

Verification may take place only in protected mode. If **VerifyAccess** is called in real mode, it returns that the memory is accessible.

Once **VerifyAccess** has verified that the process has the needed access to a specific address location, the device driver need not request access verification each time it yields the CPU during task-time processing of this process's request. If the process makes a new request, however, the device driver must request access verification.

Note also that, prior to requesting a **Lock** on user process-supplied addresses, the device driver must verify the user process's access to the memory by using the **VerifyAccess** call. The device driver must not yield the CPU between the **VerifyAccess** and the **Lock**; otherwise, the user process could shrink the segment before it has been locked. Once the user access has been verified, the device driver may convert the virtual address to a physical address and lock the memory. The access verification is then valid for the duration of the lock.

See Also:

Lock

Interrupt Handling
(Section 5.9, Pages 356-362)

EOI: Issue End-Of-Interrupt
(Page 357)

Purpose:

The **EOI** function is used to issue an End-Of-Interrupt (EOI) to the master/slave 8259 interrupt controller as appropriate to the interrupt level.

Calling Sequence:

```
MOV  AL,IRQnum              ;Interrupt level number (0–0FH)
MOV  DL,DEVHLP_EOI
CALL [Device_Help]
```

where:

IRQnum is the interrupt level number in the range from 00H to 0FH. Any software interrupt vector may be set. Invalid ranges are 08H–0FH, 50H–57H, and 70H–77H.

Returns:

None.

Remarks:

The **EOI** function is used to issue an End-Of-Interrupt to the 8259 interrupt controller on behalf of a device driver interrupt handler. If the specified interrupt level is for the slave 8259 interrupt controller, then this routine issues the EOI to both the master and slave 8259 controllers.

Device drivers must use this service in their interrupt handlers for compatibility with future versions of MS OS/2.

This function is bimodal and may be called at initialization time for interrupt processing.

EOI does not change the state of the interrupt flag.

If the device driver is going to return to the operating system immediately after issuing the **EOI**, it should disable interrupts prior to the **EOI**. This allows the processing for this interrupt level to finish before the system services the next interrupt, and reduces the probability of a stack overflow.

SetIRQ: Set Hardware Interrupt Handler
(Pages 358–359)

Purpose:

The **SetIRQ** function registers a device interrupt handler for a hardware interrupt level.

Calling Sequence:

```
MOV  AX,Handler          ;Interrupt handler offset
MOV  BX,IRQnum           ;Interrupt level number (0–0FH)
MOV  DH,SharedInt        ;Interrupt sharing
MOV  DL,DEVHLP_SETIRQ
CALL [Device_Help]
```

where:

Handler is the offset to the interrupt handler.

IRQnum is the interrupt level number in the range from 00H to 0FH. Any software interrupt vector may be set. Invalid ranges are 08H–0FH, 50H–57H, and 70H–77H.

SharedInt specifies whether interrupt sharing is available with one of the following values:

Value	Meaning
0	Not shared
1	Shared

Returns:

C —cleared if no error, set if error.

AX = Error code:

- IRQ is not available

Remarks:

The attempt to register an interrupt handler for an IRQ that is shared will fail if the IRQ is:

- Already owned by another device driver as "not shared"
- Owned by a real-mode box interrupt handler
- The IRQ used to cascade the slave 8259 interrupt controller.

The attempt to register an interrupt handler for an IRQ that is *not* shared will fail if the IRQ is:

- Already owned by another device driver as shared or not shared
- Owned by a real-mode box interrupt handler
- The IRQ used to cascade the slave 8259 interrupt controller.

SetIRQ enables the interrupt level at the 8259 interrupt controller on behalf of the device driver interrupt handler if the IRQ is available.

The **DS** register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the **DS** register, it should restore **DS** to its original value.

The device-driver interrupt handler does not need to save/restore registers on entry/exit since this function is performed by the MS OS/2 interrupt manager.

To provide improved response to asynchronous-communications interrupts, the #1 Intel 8259A Interrupt Controller is operated with IRQ 2 as lowest priority.

See Also:

PhysToVirt

UnSetIRQ: Remove Hardware Interrupt Handler
(Page 362)

Purpose:

The **UnSetIRQ** function removes the current hardware interrupt handler.

Calling Sequence:

```
MOV  BX,IRQnum              ; Interrupt level number (0–0FH)
MOV  DL,DEVHLP_UNSETIRQ
CALL [Device_Help]
```

where:

IRQnum is the interrupt level number, in the range from 00H to 0FH previously set by the **SetIRQ** call.

Returns:

None.

Remarks:

DS must point to the device driver's data segment on entry.

Timer Services
(Section 5.10, Pages 363–368)

SetTimer: Set Timer Handler
(Pages 365–366)

Purpose:

The **SetTimer** function adds a timer handler to the list of timer handlers to be called on a timer tick.

Calling Sequence:

```
MOV  AX,TimerHandler        ;Offset of timer handler.
MOV  DL,DEVHLP_SETTIMER
CALL [Device_Help]
```

where:

TimerHandler is the offset of the timer handler to be added to the list of timer handlers.

Returns:

C—cleared if no error, set if error.

AX = Error code:

- Timer handler disallowed (maximum number of handlers reached or timer handler already set).

Remarks:

A driver driver may use a timer handler to drive a non-interrupt device instead of using time-outs with the **Block** and **Run** services. This method is preferable because when measured on a character-by-character basis, **Block** and **Run** are costly, requiring one or more task switches per character I/O.

A maximum of 32 different timer handlers is available in MS OS/2.

While a timer handler is in the format of a CALL/RETURN routine (when it is finished processing, it performs a normal return), it operates in interrupt mode. The timer handler is analogous to the user timer (INT 1CH) handler.

Note

Be sure to remain in the handler for as short a period of a time as possible.

Note that the **SetTimer** function returns an error when it is called with a *TimerHandler* address that is already on its list.

The **DS** register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the **DS** register, it should restore **DS** to its original value.

Timer handlers are responsible for saving and restoring registers on entry and exit.

See Also:

Block, PhysToVirt, Run

TickCount: Modify Timer
(Pages 367–368)

Purpose:

The **TickCount** function registers a new timer handler or modifies a previously registered timer handler to be called on every *n* timer ticks instead of every timer tick.

Calling Sequence:

```
MOV  AX,TimerHandler        ;Offset to timer handler
MOV  BX,Count               ;Number of tick counts (0–0FFFFH)
MOV  DL,DEVHLP_TICKCOUNT
CALL [Device_Help]
```

where:

TimerHandler is the offset to the timer handler for which the tick count is to be modified.

Count is the number of tick counts in the range from 00H to 0FFFFH where zero means 0FFFFH+1 ticks.

Returns:

C—cleared if no error, set if error.

AX = Error code:

- Timer handler cannot be modified or set.

Remarks:

For a new timer handler, **TickCount** registers the timer handler to be called every *Count* timer ticks instead of every timer tick.

For a previously-registered timer handler, **TickCount** changes the number of ticks that must take place before the timer handler gets control. This allows device drivers to support the time-out function without needing to count timer ticks.

At task-time, this **DevHlp** may be used to modify a timerhandler registered via **SetTimer** or it may be used to register a new timer handler that is initially invoked every *Count* ticks.

In user mode (during task-time) or interrupt mode (during interrupt-time), this **DevHlp** may only be used to modify a previously-registered timer handler. This allows an interrupt handler to reset the timing condition at interrupt-time.

Note that **SetTimer** sets a default *Count* of 1. Multiple **TickCount** requests may be issued for a given timer handler, but only the last **TickCount** setting will be in effect.

TickCount affects only the specified registered timer handler. It has no effect on other timer handlers.

An error is returned via the carry flag (**CF**) if the timer handler cannot be modified or set.

The **DS** register should be set to the device driver's data segment. If the device driver has made a **PhysToVirt** call referencing the **DS** register, it should restore DS to its original value.

Timer handlers are responsible for saving and restoring registers on entry and exit.

A maximum of 32 different timer handlers is available in MS OS/2.

See Also:

PhysToVirt, SetTimer

Monitor Management
(Section 5.11, Pages 369–379)

MonFlush: Flush Data from Monitor Stream
(Pages 371–372)

Purpose:

The **MonFlush** function removes all data from the monitor stream.

Calling Sequence:

```
MOV  AX,MonitorHandle        ; MonitorCreate handle for chain
MOV  DL,DEVHLP_MONFLUSH
CALL [Device_Help]
```

where:

MonitorHandle is the **MonitorCreate** handle for the chain.

Returns:

C—cleared if no error, set if error.

AX = Error code:

- Invalid monitor handle.

Remarks:

This function may be called only at task time.

Until the flush completes, subsequent **MonWrite** requests will fail (or block).

The general format of monitor records requires that every record contain a flag word as the first entry. One of the flags indicates that this record is a flush record, which consists of the flag word only. Monitors use this flush record along the chain to reset internal state information and to ensure that all internal buffers are flushed. The flush record must be passed along to the next monitor, since the monitor dispatcher will not process any more information until the flush record is received at the end of the chain.

When this function is called, it can change the state of the interrupt flag. You should not depend on this state remaining unchanged.

See Also:

MonitorCreate, MonWrite

MonWrite: Pass Data Records to Monitors
(Pages 376–377)

Purpose:

The **MonWrite** function passes data records to the monitors for filtering.

Calling Sequence:

```
LDS  SI,DataRecord          ; Address of data record
MOV  CX,Count               ; Byte count of data record
MOV  AX,MonitorHandle       ; MonitorCreate handle for chain
MOV  DH,WaitFlag            ; Wait/No Wait flag
MOV  DL,DEVHLP_MONWRITE
CALL [Device_Help]
```

where:

All the parameters are supplied by the device driver.

The *WaitFlag* is zero if the **MonWrite** request occurs at task or user time and if the device driver wishes to have the monitor dispatcher perform the synchronization. If no wait is required, a value of one is specified. At interrupt time, it is necessary to specify a value of one.

Returns:

C—cleared if no error, and set if error.

AX = Error code:

- Invalid monitor handle.
- Not enough memory.

Remarks:

The **MonWrite** function may be called at either task or interrupt time. The "Not enough memory" condition can arise when the **MonWrite** call is made and the buffer does not contain sufficient free space to receive the data. If this condition occurs at interrupt time, it signifies an overrun. If it occurs at task (or user) time, the process can block.

A **MonFlush** call that is in progress can also cause a "not enough memory" condition. Waiting until the **MonFlush** call has completed may correct this condition.

Each call to **MonWrite** sends a set of data, which are considered a single, complete record.

When this function is called, it can change the state of the interrupt flag. You should not depend on this state remaining unchanged.

See Also:

MonFlush, MonitorCreate

System Services
(Section 5.12, Pages 380–391)

GetDosVar: Return Pointer to DOS Variable
(Pages 381–382)

Purpose:

The **GetDosVar** function returns the address of an internal DOS variable.

Calling Sequence:

```
MOV  AL,VarNumber          ;Variable wanted
MOV  DL,DEVHLP_GETDOSVAR
CALL [Device_Help]
```

where:

VarNumber is the number of the MS OS/2 variable to be returned.

Returns:

C—cleared if no error.

AX:BX points to the variable.

C—set if error.

Remarks:

The list of available variables is subject to growth in future versions of the operating system. The operating system maintains these variables for the benefit of device drivers.

The returned pointer (address) is a bimodal pointer. The address returned is that of the indicated variable, which may contain a structure or a vector to a facility.

The variables are read-only and are described as follows:

Index	Value	Comment
1	*SysINFOSeg*:WORD	Bimodal segment address of the system (GDT) INFO segment. Valid at both task time and interrupt time.
2	*LocINFOSeg*:DWORD	Selector/Segment address of the local (LDT) INFO segment. Valid only at task time.
3	Reserved	
4	*VectorSDF*:DWORD	Vector to the stand alone dump facility. Valid at both task time and interrupt time.
5	*VectorReboot*:DWORD	Vector to reboot the DOS. Valid at both task time and interrupt time.
6	*VectorMSATS*:QWORD	Vectors to the MSATS facility: protected mode and real 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	*CodePage Tag Ptr*.DWORD	Segment/offset of the current code page of the real-mode (DOS 3.x) box. Valid only at task time.

See Also:

TCYield, Yield

SendEvent: Indicate Event
(Pages 383–384)

Purpose:

The **SendEvent** function is called by a device driver to indicate the occurrence of an event.

Calling Sequence:

```
MOV  AH,Event                ;Event being signaled
MOV  BX,Argument             ;Parameter for event being signaled
MOV  DL,DEVHLP_SENDEVENT
CALL [Device_Help]
```

where:

Event identifies the event being signaled. It has one of the following values:

Value	Meaning
0	Session manager hot key from the mouse *Argument* is a two-byte time stamp where the high byte is seconds and the low byte is hundredths of seconds.
1	CONTROL-BREAK *Argument* is reserved and must be zero.
2	CONTROL-C *Argument* is reserved and must be zero.
3	CONTROL-SCROLL LOCK *Argument* is reserved and must be zero.
4	CONTROL-PRINTSCREEN *Argument* is reserved and must be zero.
5	SHIFT-PRINTSCREEN *Argument* is reserved and must be zero.
6	Session manager hot key from the keyboard *Argument* is the *Hot key ID* defined via the keyboard IOCtl, **Function 56H**, Set Session Manager Hot Key.

Argument is the parameter for the event being signaled.

Returns:

C—set if error, and cleared if no error.

Remarks:

None.