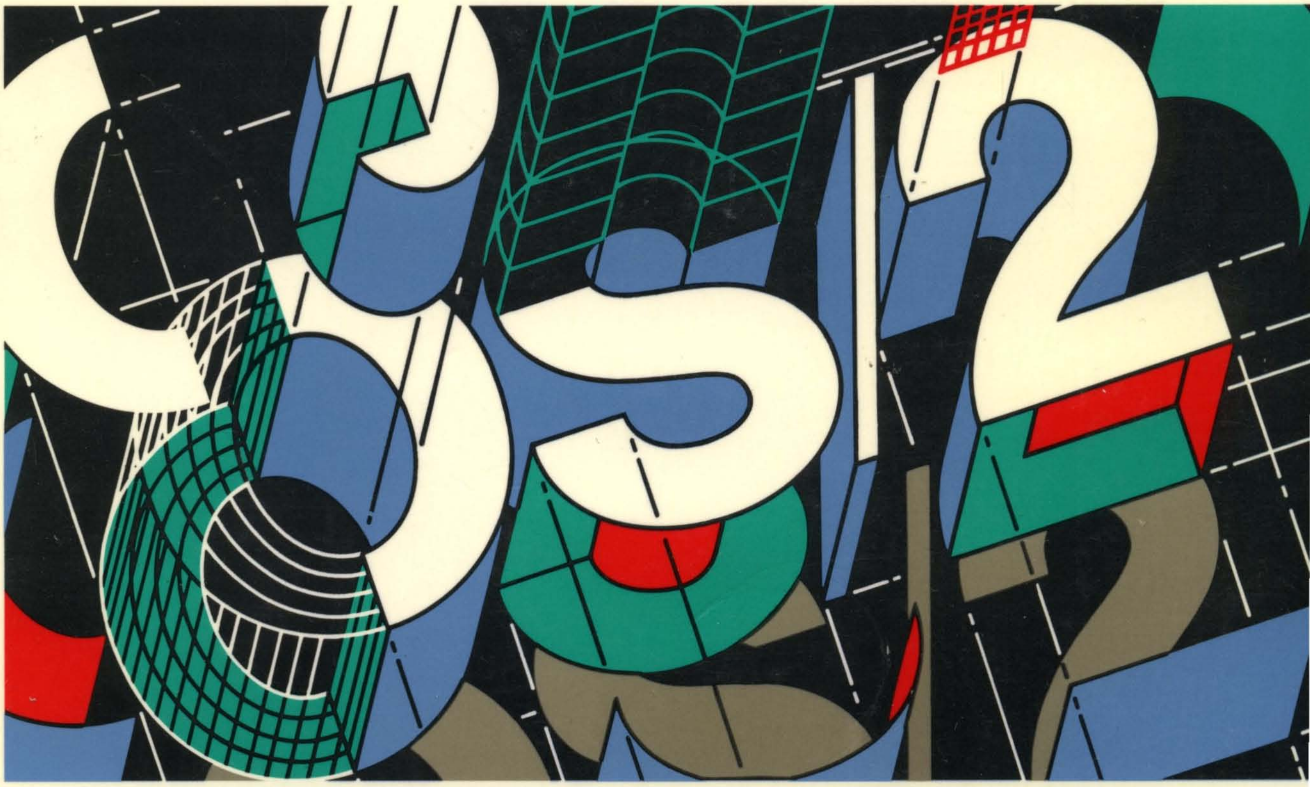
Volume 3

M I C R O S O F T®

OS/2 Programmer's Reference



Microsoft OS|2

PROGRAMMER'S REFERENCE LIBRARY

Volume
3

Microsoft® Operating System/2 Programmer's Reference

Version 1.1

Written, edited, and produced
by Microsoft Corporation
Distributed by Microsoft Press

Microsoft
OS|2
PROGRAMMER'S
REFERENCE
LIBRARY

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
16011 NE 36th Way, Box 97017, Redmond, Washington 98073-9717

Library of Congress Cataloging in Publication Data
Microsoft OS/2 programmer's reference.
Includes index.
1. Microsoft OS/2 (Computer operating system) I. Microsoft Press
QA76.76.063078 1989 005.4'469 89-2817
ISBN 1-55615-222-1(Vol. 3)

Printed and bound in the United States of America.
1 2 3 4 5 6 7 8 9 FGFG 3 2 1 0 9

Distributed to the book trade in the United States by Harper & Row.
Distributed to the book trade in Canada by General Publishing Company, Ltd.
Distributed to the book trade outside the United States and Canada
by Penguin Books Ltd.
Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

Document No. LN0702C-110-R00-0289

Contents

Appendix C Country and Code-Page Information

Figures

Tables

Chapter 1

Introduction

1.1 Overview

This manual describes the **Dos**, **Kbd**, **Mou**, and **Vio** system functions of Microsoft® Operating System/2 (MS® OS/2). These functions, also called the base system functions, let MS OS/2 programs use the operating system to carry out tasks such as reading from and writing to disk files; allocating memory; starting other programs; and using the keyboard, mouse, and video screen.

MS OS/2 system functions are designed to be used in C, Pascal, and other high-level-language programs, as well as in assembly-language programs. In MS OS/2, all programs request operating-system services by calling system functions.

This chapter, "Introduction," shows how to use this manual, provides a brief description of MS OS/2 calling conventions, illustrates function calls in various languages, and outlines MS OS/2 naming conventions.

Chapter 2, "Functions Directory," is an alphabetical listing of MS OS/2 base system functions. This chapter defines each function's purpose, gives its syntax, describes the function parameters, and gives possible return values. Many functions also show simple program examples that illustrate how the function is used to carry out simple tasks.

Chapter 3, "Input-and-Output Control Functions," lists the input-and-output control (**IOCtl**) functions used to control input and output devices such as serial ports, the keyboard, and the mouse.

Chapter 4, "Types, Macros, Structures," describes the types, macros, and structures used by MS OS/2 base system functions.

Chapter 5, "File Formats," describes the format of files and other large data structures used by MS OS/2 base system functions. These formats include keyboard translation tables and video I/O fonts.

Appendix A, "Error Values," lists error codes and their corresponding values.

Appendix B, "ANSI Escape Sequences," lists the escape sequences used by MS OS/2.

Appendix C, "Country and Code-Page Information," lists information contained in the country and code-page files used by MS OS/2 system functions. This includes code-page tables, code-page identifiers, and country-specific information.

This manual is intended to fully describe MS OS/2 base system functions and the structures and file formats used with these functions. It does not show how to use these functions to carry out specific tasks. For more information on this topic, see the *Microsoft Operating System/2 Programmer's Reference, Volume 1.* Also, this manual does not describe MS OS/2 Presentation Manager functions. Presentation Manager functions let programs use the window-management and graphics features of MS OS/2. For more information on MS OS/2 Presentation Manager functions, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

1.2 How to Use This Manual

This manual provides detailed information about each MS OS/2 base system function, macro, and structure. Each description has the following format:

Figure 1.1
Sample Reference Page

❶ ■ **DosBeep**

❷
```
USHORT DosBeep(usFrequency, usDuration)
USHORT usFrequency;    /* frequency in hertz       */
USHORT usDuration;     /* duration in milliseconds */
```

❸ The **DosBeep** function generates sound from the speaker.

❹ The **DosBeep** function is a family API function.

❺ **Parameters**

usFrequency Specifies the frequency of the sound in hertz (cycles-per-second). This parameter can be any value from 0x0025 through 0x7FFF.

usDuration Specifies the length of the sound in milliseconds.

❻ **Return Value**

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_FREQUENCY

❼ **Example**

This example calls the **DosBeep** function and produces audible tones:

```
SHORT i;
for (i = 0; i < 10; i++) {
    DosBeep(600, 175);
    DosBeep(1200, 175);
}
```

❽ **See Also**

WinAlarm

These are the elements shown:

1 The function, macro, or structure name.

2 The function, macro, or structure syntax. The syntax specifies the number of parameters (or fields) and gives the type of each. It also gives the order (from left to right) that parameters must be pushed on the stack. Comments to the right briefly describe the purpose of the parameter.

3 A description of the function, macro, or structure, including its purpose and details of operation.

4 Any special consideration for the function, including whether a function can be used in family API programs.

5 A full description of each parameter (or field), including permitted values and related structures.

6 A description of the function return value, including possible error values.

7 An example showing how the function can be used to accomplish a simple task.

8 A list of related functions and structures.

1.2.1 C Format

In this manual, the syntax for MS OS/2 functions is given in C-language format. In your C-language sources, the function name must be spelled exactly as given in the syntax, and the parameters must be used in the order given in the syntax. This syntax also applies to Pascal program sources.

The following example shows how to call the **DosBeep** function in a C-language program:

```
/* play a note for 1 second */

DosBeep(660,                /* 660 cycles-per-second      */
        1000);              /* play for 1000 milliseconds */
```

1.2.2 MS OS/2 Include Files

This manual uses many types, structures, and constants that are not part of standard C language. These items, designed for MS OS/2, are defined in the MS OS/2 C-language include files provided with the Microsoft OS/2 Presentation Manager Softset and the Microsoft OS/2 Presentation Manager Toolkit.

In C-language programs, the **#include** directive specifying *os2.h*, the MS OS/2 C-language include file, can be placed at the beginning of the source file to include the definitions for the special types, structures, and constants. Although there are many MS OS/2 include files, the *os2.h* file contains the additional **#include** directives needed to process the basic MS OS/2 definitions.

To speed up processing of the MS OS/2 C-language include files, many definitions are processed only if the C-language program explicitly defines a corresponding include constant. An include constant is simply a constant name, with the prefix INCL_, that controls a portion of the include files. If a constant is defined using the **#define** directive, the corresponding MS OS/2 definitions are processed. For a list of the include constants and a description of the MS OS/2 system functions they enable, see the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

1.2.3 MS OS/2 Calling Conventions

You must know MS OS/2 calling conventions to use MS OS/2 functions in other high-level languages or in assembly language. MS OS/2 functions use the Pascal (sometimes called the PLM) calling convention for passing parameters, and they apply some additional rules to support dynamic-link libraries. The following rules apply:

- You must push the parameters on the stack. In this manual, each function description lists the parameters in the order they must be pushed. The left parameter must be pushed first, the right parameter last. If a parameter specifies an address, the address must be a far address; that is, it must have the form *selector*:*offset*. The *selector* must be pushed first, then the *offset*.
- The function automatically removes the parameters from the stack as it returns. This means the function must have a fixed number of parameters.

- You must use an intersegment call instruction to call the function. This is required for all dynamic-link-library functions.
- The function returns a value, possibly an error value, in either the **ax** register or the **dx:ax** register pair. Only the **di** and **si** register values are guaranteed to be preserved by the function. MS OS/2 base system functions may preserve other registers as well, but they do not preserve the **flags** register. The contents of the **flags** register are undefined; specifically, the direction flag in the register may be changed. However, if the direction flag was zero before the function was called, it will be zero after the function returns.

The following example shows how MS OS/2 calling conventions apply to the **DosOpen** function in an assembly-language program:

```
EXTRN DOSOPEN:FAR
name            db      "abc", 0
hFile           dw      0
usAction        dw      0

push    ds                      ; filename to open
push    offset name
push    ds                      ; address of file handle
push    offset hFile
push    ds                      ; address to store action taken
push    offset usAction
push    0                       ; size of new file 0100H
push    100
push    0                       ; file's attribute
push    0010H                   ; create file if it does not exist
push    0041H                   ; open file for writing, share with all
push    0                       ; reserved
push    0
call    DOSOPEN
```

The following example shows how to call the same **DosOpen** function in a C-language program. In C, the **DosOpen** function name, parameter types, and constant names are defined in *os2.h*, the MS OS/2 C-language include file.

```
# include <os2.h>

HFILE hfile;
USHORT usAction;

DosOpen("abc",          /* filename to open              */
    &hfile,             /* address of file handle        */
    &usAction,          /* address to store action taken */
    100L,               /* size of new file              */
    FILE_NORMAL,        /* file's attribute              */
    FILE_CREATE,        /* create file if it does not exist */
    OPEN_SHARE_DENYNONE |   /* share with all            */
    OPEN_ACCESS_WRITEONLY,  /* open for writing          */
    0L);                    /* reserved                  */
```

1.2.4 Bit Masks in Function Parameters

Many MS OS/2 system functions accept or return bit masks as part of their operation. A bit mask is a collection of two or more bit fields within a single byte, or a short or long value. Bit masks provide a way to pack many Boolean flags (flags whose values represent on/off or true/false values) into a single parameter or structure field. In assembly-language programming, it is easy to individually set, clear, or test the bits in a bit mask by using instructions that modify or examine bits within a byte or a word. In C-language programming, however, the programmer does not have direct access to these instructions, so the bitwise AND and OR operators typically are used to examine and modify the bit masks.

Since this manual presents the syntax of MS OS/2 system functions in C-language syntax, it also defines bit masks in a way that is easiest to work with using the C language: as a set of constant values. When a function parameter is a bit mask, this manual provides a list of constants (named or numeric) that represent the correct values used to set, clear, or examine each field in the bit mask. For example, the **fbType** field of the **VIOMODEINFO** structure in the **VioSetMode** function specifies three values: VGMT_OTHER, VGMT_GRAPHICS, and VGMT_DISABLEBURST. These represent the "set" values of the first three fields in the bit mask. Typically, the description associated with the value explains the result of the function if the given value is used; that is, when the corresponding bit is set. Generally, the opposite result is assumed when the value is not used. For example, using VGMT_GRAPHICS in the **fbType** field enables graphics mode; not using it disables graphics mode.

1.2.5 Structures

Many MS OS/2 system functions use structures as input and output parameters. This manual defines all structures and their fields using C-language syntax. In most cases, the structure definition presented is copied directly from the C-language include files provided with the Microsoft C Optimizing Compiler. Occasionally, an MS OS/2 function may have a structure that has no corresponding include-file definition. In such cases, this manual gives an incomplete form of the C-language structure definition to indicate that the structure is not already defined in an include file.

1.3 Naming Conventions

In this manual, all parameter, variable, structure, field, and constant names conform to MS OS/2 naming conventions. MS OS/2 naming conventions are rules that define how to create names that indicate both the purpose and data type of an item used with MS OS/2 system functions. These naming conventions are used in this manual to help you readily identify the purpose and type of the function parameters and structure fields. These conventions are also used in most MS OS/2 sample program sources to make the sources more readable and informative.

1.3.1 Parameter and Field Names

With MS OS/2 naming conventions, all parameter and field names consist of up to three elements: a prefix, a base type, and a qualifier. A name always consists of at least a base type or a qualifier. In most cases, the name also includes a prefix.

The base type, always written in lowercase letters, identifies the data type of the item. The prefix, also written in lowercase letters, specifies additional information about the item, such as whether it is a pointer, an array, or a count of bytes. The qualifier, a short word or phrase written with the first letter of each word uppercase, specifies the purpose of the item.

There are several standard prefixes and base types. These are used for the data types most frequently used with MS OS/2.

1.3.1.1 Prefixes

The following is a list of standard prefixes used in MS OS/2 naming conventions:

Prefix	Description
p	Pointer. This prefix identifies a far, or 32-bit, pointer to a given item. For example, *pch* is a far pointer to a character.
np	Near pointer. This prefix identifies a near, or 16-bit, pointer to a given item. For example, *npch* is a near pointer to a character.
a	Array. This prefix identifies an array of two or more items of a given type. For example, *ach* is an array of characters.
i	Index. This prefix identifies an index into an array. For example, *ich* is an index to one character in an array of characters.
c	Count. This prefix identifies a count of items. It is usually combined with the base type of the items being counted instead of the base type of the actual parameter. For example, *cch* is a count of characters even though it may be declared with the type **USHORT**.
h	Handle. This prefix is used for values that uniquely identify an object but that cannot be used to access the object directly. For example, *hfile* is a handle of a file.
off	Offset. This prefix is used for values that represent offsets from the beginning of a buffer or a structure. For example, *off* is the offset from the beginning of the given segment to the specified byte.
id	Identifier. This prefix is used for values that identify an object. For example, *idSession* is a session identifier.

1.3.1.2 Base Types

The following is a list of standard base types used in MS OS/2 naming conventions:

Base type	Type/Description
f	**BOOL**. A 16-bit flag or Boolean value. The qualifier should describe the condition associated with the flag when it is TRUE. For example, *fSuccess* is TRUE if successful, FALSE if not; *fError* is TRUE if an error occurs and FALSE if no error occurs. For objects of type **BOOL**, a zero value implies FALSE; a nonzero value implies TRUE.
ch	**CHAR**. An 8-bit signed value.
s	**SHORT**. A 16-bit signed value.
l	**LONG**. A 32-bit signed value.
uch	**UCHAR**. An 8-bit unsigned value.
us	**USHORT**. A 16-bit unsigned value.
ul	**ULONG**. A 32-bit unsigned value.
b	**BYTE**. An 8-bit unsigned value. Same as *uch*.
sz	**CHAR[ ]**. Array of characters, terminated with a null character (the last byte is set to zero).
fb	**UCHAR**. Array of flags in a byte. This base type is used when more than one flag is packed in an 8-bit value. Values for such an array are typically created by using the logical OR operator to combine two or more values.
fs	**USHORT**. Array of flags in a short (16-bit unsigned value). This base type is used when more than one flag is packed in a 16-bit value. Values for such an array are typically created by using the logical OR operator to combine two or more values.
fl	**ULONG**. Array of flags in a long (32-bit unsigned value). This base type is used when more than one flag is packed in a 32-bit value. Values for such an array are typically created by using the logical OR operator to combine two or more values.
sel	**SEL**. A 16-bit value used to hold a segment selector.

The base type for a structure is usually derived from the structure name. An MS OS/2 structure name, always written in uppercase letters, is a word or phrase that describes the size, purpose, and/or intended content associated with the type. The base type is typically an abbreviation of the structure name. The following list gives the base types for the structures described in this manual:

ctryc	*kbdtyp*	*ptbuf*
ctryi	*lnctl*	*driv*
date	*lis*	*qresc*
dcbinf	*mdmst*	*resc*
trckl	*mnin*	*shftst*
bspblk	*mnout*	*kbsi*
fdate	*mouev*	*htky*
scrgrp	*moupl*	*stdata*
findbuf	*moups*	*mnpos*
flock	*mouqi*	*stsdata*
frm	*mourt*	*rtdly*
fsalloc	*mousc*	*vioci*
fsinf	*trckfmt*	*viofi*
dosfsrs	*mxs*	*vioin*
fsts	*mxsl*	*vioint*
ftime	*rxq*	*viomi*
gis	*dvpblck*	*vioos*
htype	*pidi*	*viopal*
kbci	*nmpinf*	*viopb*
kbstkbs	*pi*	*vol*
kbxl	*ptrdfnc*	

1.3.2 Constant Names

A constant name is a descriptive name for a numeric value used with an MS OS/2 function. All constant names are written in uppercase letters and have a prefix derived from the name of the function, object, or idea associated with the constant. The prefix is followed by an underscore (_) and the rest of the constant name, which indicates the meaning of the constant and may specify a value, action, color, or condition. A few common constants do not have prefixes—for example, NULL is used for null pointers of all types, and TRUE and FALSE are used with the **BOOL** data type.

1.4 Notational Conventions

The following notational conventions are used throughout this manual:

Convention	Meaning
bold	Bold type is used for keywords—for example, the names of functions, data types, structures, and macros. These names are spelled exactly as they should appear in source programs.
italics	Italic type is used to indicate the name of an argument; this name must be replaced by an actual argument. Italics are also used to show emphasis in text.
`monospace`	Monospace type is used for example program-code fragments.

Chapter

2

Functions Directory

2.1 Introduction

This chapter describes MS OS/2 **Dos**, **Kbd**, **Mou**, and **Vio** functions. These functions, also called MS OS/2 base system functions, provide the support programs need to access the basic operating-system features of MS OS/2, such as multitasking, memory management, and input and output. The **Dos**, **Kbd**, **Mou**, and **Vio** functions represent four distinct function groups. As described in the following list, programs use these function groups to carry out specific tasks:

Function group	Usage
Dos	Use the disk operating system (**Dos**) functions in full-screen and Presentation Manager sessions to read from and write to disk files, to allocate memory, to start threads and processes, to communicate with other processes, and to access your computer's devices directly. Most functions in this group can be used in Presentation Manager applications.
Kbd	Use the keyboard (**Kbd**) functions in full-screen sessions to read keystrokes from the keyboard, to manage multiple logical keyboards, and to change code pages and translation tables. Since the Presentation Manager session provides its own keyboard support, **Kbd** functions are not needed in Presentation Manager applications.
Mou	Use the mouse (**Mou**) functions in full-screen sessions to read mouse input from the mouse-event queue, to set the mouse-pointer shape, and to manage the mouse for all processes in a session. As with the keyboard, the Presentation Manager session provides its own mouse support, so **Mou** functions are not needed in Presentation Manager applications.
Vio	Use the video input-and-output (**Vio**) functions in full-screen sessions to write characters and character attributes to the screen, to create pop-up windows for messages, to change the video modes, and to access physical video memory. **Vio** functions can also be used in advanced video-input-and-output (AVIO) applications for the Presentation Manager session to write characters and character attributes in a window. Most Presentation Manager applications, however, use the graphics programming interface (Gpi) to write text in a window.

Many functions in this chapter are also family API functions. This means they can be used in dual-mode programs—that is, programs that run in either MS OS/2 or MS-DOS®. The family API functions are clearly marked.

In this chapter, complete syntax, purpose, and parameter descriptions are given for each function. Types, macros, and structures used by a function are given with the function; these are defined more fully in Chapter 4, "Types, Macros, Structures." The numeric values for error values returned by the functions are listed in Appendix A, "Error Values."

Many of the function descriptions in this chapter include examples. The examples show how to use the functions to accomplish simple tasks. In nearly all cases, the examples are code fragments, not complete programs. A code fragment is intended to show the context in which a function can be used, but often assumes that variables, structures, and constants used in the example have been defined and/or initialized. Also, a code fragment may use comments to represent a task instead of giving the actual statements.

Although the examples are not complete, you can still use them in your programs if you take the following steps:

- Include the *os2.h* file in your program.
- Define the appropriate include constants for the functions, structures, and constants used in the example.
- Define and initialize all variables.
- Replace comments that represent tasks with appropriate statements.
- Check return values for errors and take appropriate actions.

2.2 Functions

The following is a complete list, in alphabetical order, of the MS OS/2 **Dos**, **Kbd**, **Mou**, and **Vio** functions.

■ DosAllocHuge

```
USHORT DosAllocHuge(usNumSeg, usPartialSeg, psel, usMaxNumSeg, fsAlloc)
USHORT usNumSeg;        /* number of segments to allocate                */
USHORT usPartialSeg;    /* number of bytes in last segment               */
PSEL psel;              /* pointer to variable for selector allocated    */
USHORT usMaxNumSeg;     /* maximum number of segments to reallocate      */
USHORT fsAlloc;         /* sharable/discardable flags                    */
```

The **DosAllocHuge** function allocates a huge memory block. This block consists of one or more 65,536-byte memory segments and one additional segment of the size specified by the *usPartialSeg* parameter.

The **DosAllocHuge** function allocates the segments and copies the selector of the first segment to the variable pointed to by the *psel* parameter. Selectors for the remaining segments are consecutive and must be computed by using the selector offset.

The **DosAllocHuge** function can specify that segments be sharable or discardable. If the process that calls **DosAllocHuge** specifies that the segments can be shared, then it can call the **DosGiveSeg** function to make the location or the allocated segments available to another process. The other process must use the **DosGetSeg** function to access the shared memory. For more information about sharable and discardable segments, see the "Comments" section under the **DosAllocSeg** function.

The **DosAllocHuge** function is a family API function.

Parameters

usNumSeg Specifies the number of 65,536-byte segments to be allocated.

usPartialSeg Specifies the number of bytes in the last segment. This number can be any value from 0 through 65,535. If it is zero, no additional segment is allocated.

psel Points to the variable that receives the selector of the first segment.

usMaxNumSeg Specifies the maximum number of segments that can be specified in any subsequent call to the **DosReallocHuge** function. If the *usMaxNumSeg* parameter is zero, the memory cannot be reallocated to a size greater than its original size, but it can be reallocated to a smaller size.

fsAlloc Specifies whether the segments can be shared with other processes or can be discarded. The *fsAlloc* parameter can be one or more of the following values:

Value	Meaning
SEG_DISCARDABLE	Create discardable segments.
SEG_GETTABLE	Create sharable segments that other processes can retrieve by using the **DosGetSeg** function.
SEG_GIVEABLE	Create sharable segments that the owning process can give to other processes by using the **DosGiveSeg** function.
SEG_NONSHARED	Create nonsharable, nondiscardable segments. This value cannot be combined with any other value.

If the shared or discardable attributes are not specified, only the process that creates the segment can access it, and the contents of the segment remain in memory until the process frees the segment.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_NOT_ENOUGH_MEMORY

Comments

Each segment in the huge memory block has a unique selector. The selectors are consecutive. The *psel* parameter specifies the value of the first selector; the remaining selectors can be computed by adding the selector offset to the first selector one or more times—that is, once for the second selector, twice for the third, and so on. The selector offset is a multiple of 2, as specified by the shift count retrieved by using the **DosGetHugeShift** function. For example, if the shift count is 2, the selector offset is 4 (1 << 2). If the selector offset is 4 and the first selector is 6, then the second selector is 10, the third is 14, and so on.

The system may move or swap the memory segments as directed by the **memman** command in the *config.sys* file. Moving and swapping have no effect on the value of the segment selectors, so you can compute the selectors at any time and save them; they will remain available for use as long as the memory remains allocated.

The **DosAllocHugeSeg** function automatically locks the segment. A locked segment cannot be discarded. You must use the **DosUnlockSeg** function to unlock the segment and permit discarding. To prevent the memory manager from discarding an unlocked discardable segment, use the **DosLockSeg** function.

The **DosFreeSeg** function frees all segments if you pass it the first selector.

Restrictions

In real mode, the following restrictions apply to the **DosAllocHuge** function:

- The *usPartialSeg* parameter is rounded up to the next paragraph (16-byte) value.
- The actual segment address is copied to the *psel* parameter.

Example

This example calls the **DosAllocHuge** function to allocate two segments with 64K and one segment with 200 bytes. It then converts the first selector to a huge pointer that can access all the memory allocated.

```
CHAR huge *pchBuffer;
SEL sel;
DosAllocHuge(3,                     /* number of segments            */
    200,                            /* size of last segment          */
    &sel,                           /* address of selector           */
    5,                              /* maximum segments for reallocation */
    SEG_NONSHARED);                 /* sharing flag                  */
pchBuffer = MAKEP(sel, 0);          /* converts to a pointer         */
```

See Also

DosAllocSeg, DosFreeSeg, DosGetHugeShift, DosGetSeg, DosGiveSeg, DosLockSeg, DosReallocHuge, DosUnlockSeg

DosAllocSeg

USHORT DosAllocSeg(*usSize*, *psel*, *fsAlloc***)**

```
USHORT usSize;    /* number of bytes requested                  */
PSEL psel;        /* pointer to variable for selector allocated */
USHORT fsAlloc;   /* sharable/discardable flags                 */
```

The **DosAllocSeg** function allocates a memory segment and copies the segment selector to the variable pointed to by the *psel* parameter. The segment can have from 1 through 65,536 bytes.

The **DosAllocSeg** function can specify that the segment be sharable or discardable. If the process that calls **DosAllocSeg** specifies that the segments can be shared, then it can call the **DosGiveSeg** function to make the location or the allocated segments available to another process. The other process must use the **DosGetSeg** function to access the shared memory.

The **DosAllocSeg** function is a family API function.

Parameters

usSize Specifies the number of bytes to be allocated. This number can be any value from 0 through 65,535. If it is zero, the function allocates 65,536 bytes.

psel Points to the variable that receives the segment selector.

fsAlloc Specifies whether the segment can be shared with other processes or can be discarded. The *fsAlloc* parameter can be one or more of the following values:

Value	Meaning
SEG_DISCARDABLE	Create a discardable segment.
SEG_GETTABLE	Create a sharable segment that other processes can retrieve by using the **DosGetSeg** function.
SEG_GIVEABLE	Create a sharable segment that the owning process can give to other processes by using the **DosGiveSeg** function.
SEG_NONSHARED	Create a nonsharable, nondiscardable segment. This value cannot be combined with any other value.

If the sharable or discardable attributes are not specified, only the process that creates the segment can access it, and the contents of the segment remain in memory until the process frees the segment.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_NOT_ENOUGH_MEMORY

Comments

The system may move or swap the memory segment as directed by the **memman** command in the *config.sys* file. Moving and swapping have no effect on the segment selectors.

A sharable segment is available to the process that created it and to other processes. If necessary, the system will discard an unlocked discardable segment in order to satisfy another allocation request. The new allocation request can come from any process, including the process that allocated the segment being discarded.

Discardable segments are useful for holding information that is accessed for short periods of time and that can be regenerated quickly if discarded. Examples are cache buffers for a database package, saved bitmap images for obscured windows, and precomputed display images for a word-processing application. Although the data in the segment is lost when the segment is discarded, the segment can be restored to its original size by using the **DosReallocSeg** function.

The **DosAllocSeg** function automatically locks the segment. A locked segment cannot be discarded. You must use the **DosUnlockSeg** function to unlock the segment and permit discarding. To prevent the memory manager from discarding an unlocked discardable segment, use the **DosLockSeg** function.

The **DosFreeSeg** function frees the segment.

Restrictions

In real mode, the following restrictions apply to the **DosAllocSeg** function:

- The *usSize* parameter is rounded up to the next paragraph (16-byte) value.
- The actual segment address is copied to the *psel* parameter.

Example

This example calls the **DosAllocSeg** function to allocate 26,953 bytes. It then converts the selector to a far pointer that can access the allocated bytes.

```
PCH pchBuffer;
SEL sel;

DosAllocSeg(26953,                  /* bytes to allocate      */
    &sel,                           /* address of selector    */
    SEG_NONSHARED);                 /* sharing flag           */
pchBuffer = MAKEP(sel, 0);          /* converts to a pointer */
```

See Also

DosAllocHuge, DosFreeSeg, DosGetSeg, DosGiveSeg, DosLockSeg, DosReallocSeg, DosUnlockSeg

DosAllocShrSeg

```
USHORT DosAllocShrSeg(usSize, pszSegName, psel)
USHORT usSize;       /* number of bytes requested            */
PSZ pszSegName;      /* pointer to segment name              */
PSEL psel;           /* pointer to variable for selector allocated */
```

The **DosAllocShrSeg** function allocates a shared memory segment and copies the segment selector to the variable pointed to by the *psel* parameter. The segment can have from 1 through 65,536 bytes.

A shared segment can be accessed by any process that knows the segment name. A process can retrieve a selector for the segment by specifying the name in a call to the **DosGetShrSeg** function. (Shared segments allocated by using the **DosAllocSeg** function must be explicitly given and retrieved by using the **DosGiveSeg** and **DosGetSeg** functions.)

Parameters

usSize Specifies the number of bytes to be allocated. This number can be any value from 0 through 65,535. If it is zero, the function allocates 65,536 bytes.

pszSegName Points to a null-terminated string that identifies the shared memory segment. The string must have the following form:

\sharemem*name*

The segment name, *name*, must have the same format as an MS OS/2 filename and must be unique. For example, the name **\sharemem**public.dat is acceptable.

psel Points to the variable that receives the segment selector.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ALREADY_EXISTS
ERROR_INVALID_HANDLE
ERROR_NOT_ENOUGH_MEMORY

Comments

A process may allocate up to 30 shared segments.

The system may move or swap the memory segments as directed by the **memman** command in the *config.sys* file. Moving and swapping have no effect on the value of the segment selector.

The **DosFreeSeg** function frees a shared segment.

Example

This example calls the **DosAllocShrSeg** function to allocate 26,953 bytes. It gives the memory the name "\sharemem\abc.mem" so that other processes may use the memory if they know the name.

```
SEL sel;

DosAllocShrSeg(26953,                  /* bytes to allocate   */
    "\\sharemem\\abc.mem",             /* memory name         */
    &sel);                             /* address of selector */
```

See Also

DosAllocHuge, DosAllocSeg, DosFreeSeg, DosGetSeg, DosGetShrSeg, DosGiveSeg

■ DosBeep

USHORT DosBeep(*usFrequency, usDuration***)**
USHORT *usFrequency*;   /* frequency in hertz */
USHORT *usDuration*;   /* duration in milliseconds */

The **DosBeep** function generates sound from the speaker.

The **DosBeep** function is a family API function.

Parameters

usFrequency Specifies the frequency of the sound in hertz (cycles-per-second). This parameter can be any value from 0x0025 through 0x7FFF.

usDuration Specifies the length of the sound in milliseconds.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_FREQUENCY

Example

This example calls the **DosBeep** function and produces audible tones:

```
SHORT i;
for (i = 0; i < 10; i++) {
    DosBeep(600, 175);
    DosBeep(1200, 175);
}
```

See Also

WinAlarm

■ DosBufReset

USHORT DosBufReset(*hf***)**
HFILE *hf*;   /* file handle */

The **DosBufReset** function flushes the file buffers for the specified file by writing the current contents of the file buffer to the corresponding device. If the file is a disk file, the function writes to the disk and updates the directory information for the file.

Although **DosBufReset** flushes and updates information as if the file were closed, the file remains open.

The **DosBufReset** function is a family API function.

Parameters

hf Identifies the file whose buffers are flushed. This handle must have been created previously by using the **DosOpen** function. If this parameter is set to 0xFFFF, the function flushes buffers for all currently open files.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_FILE_NOT_FOUND
ERROR_INVALID_HANDLE

Comments

If the process has several open files on removeable disks, the function may have the effect of requiring the user to repeatedly swap disks.

Example

This example opens the file *abc* and writes the contents of the *abBuf* buffer to the file. It then writes the data to the disk by calling the **DosBufReset** function to flush the buffers.

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesWritten, usError;
usError = DosOpen("abc", &hf, &usAction, OL, FILE_NORMAL,
    FILE_CREATE | FILE_OPEN,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, OL);
if (!usError) {
    DosWrite(hf, abBuf, sizeof(abBuf), &cbBytesWritten);
    DosBufReset(hf);    /* flush the buffers */
```

See Also

DosClose, DosOpen, DosWrite

DosCallback

VOID DosCallback(*pfn*)
PFN *pfn*;   /* pointer to ring-3 function */

The **DosCallback** function allows a process with ring-2 input/output privilege to call a ring-3 function.

Parameters

pfn Points to the ring-3 function to be called.

Return Value

This function does not return a value.

Comments

When a process with ring-2 input/output privileges uses the **DosCallback** function to call a ring-3 function, the target function executes at ring 3 and returns to the ring-2 calling process. The ring-3 function need not conform to the ring-2 privilege level. The ring-3 function that is called by the **DosCallback** function may call a ring-2 segment before it returns.

All registers except **FLAGS** will be passed intact across this call/return sequence and may be used to pass parameters or data back and forth between rings 2 and 3. Any addresses passed from ring 2 to ring 3 must be based on ring-3 selectors, because ring-3 code cannot address ring-2 data selectors.

A ring-2 stack cannot be used to pass data to a ring-3 function.

The following **Dos** functions are valid when issued from ring 2:

DosAllocHuge	**DosGetHugeShift**	**DosReadAsync**
DosAllocSeg	**DosGetInfoSeg**	**DosReallocHuge**
DosAllocShrSeg	**DosGetMachineMode**	**DosReallocSeg**
DosBeep	**DosGetModHandle**	**DosResumeThread**
DosBufReset	**DosGetModName**	**DosRmDir**
DosCallback	**DosGetPID**	**DosScanEnv**
DosChDir	**DosGetPPID**	**DosSearchPath**
DosChgFilePtr	**DosGetProcAddr**	**DosSelectDisk**
DosCliAccess	**DosGetPrty**	**DosSemClear**
DosClose	**DosGetResource**	**DosSemRequest**
DosCloseSem	**DosGetSeg**	**DosSemSet**
DosCreateCSAlias	**DosGetShrSeg**	**DosSemSetWait**
DosCreateSem	**DosGetVersion**	**DosSemWait**
DosCreateThread	**DosGiveSeg**	**DosSendSignal**
DosCwait	**DosHoldSignal**	**DosSetCp**
DosDelete	**DosKillProcess**	**DosSetDateTime**
DosDevConfig	**DosLoadModule**	**DosSetFHandState**
DosDevIOCtl	**DosLockSeg**	**DosSetFileInfo**
DosDupHandle	**DosMakePipe**	**DosSetFileMode**
DosEnterCritSec	**DosMemAvail**	**DosSetFSInfo**
DosErrClass	**DosMkDir**	**DosSetMaxFH**
DosError	**DosMove**	**DosSetPrty**
DosExecPgm	**DosMuxSemWait**	**DosSetSigHandler**
DosExit	**DosNewSize**	**DosSetVec**
DosExitCritSec	**DosOpen**	**DosSetVerify**
DosExitList	**DosOpenSem**	**DosSizeSeg**
DosFileLocks	**DosPhysicalDisk**	**DosSleep**
DosFindClose	**DosPortAccess**	**DosSubAlloc**
DosFindFirst	**DosQAppType**	**DosSubFree**
DosFindNext	**DosQCurDir**	**DosSubSet**
DosFlagProcess	**DosQCurDisk**	**DosSuspendThread**
DosFreeModule	**DosQFHandState**	**DosTimerAsync**
DosFreeSeg	**DosQFileInfo**	**DosTimerStart**
DosFSRamSemClear	**DosQFileMode**	**DosTimerStop**
DosFSRamSemRequest	**DosQFSInfo**	**DosUnlockSeg**
DosGetCp	**DosQHandType**	**DosWrite**
DosGetDateTime	**DosQVerify**	**DosWriteAsync**
DosGetEnv	**DosRead**	

■ DosCallNmPipe

USHORT DosCallNmPipe (*pszName*, *pbInBuf*, *cbInBuf*, *pbOutBuf*, *cbOutBuf*, *pcbRead*, *ulTimeOut***)**

```
PSZ pszName;         /* pointer to pipe name              */
PBYTE pbInBuf;       /* pointer to input buffer           */
USHORT cbInBuf;      /* number of bytes in input buffer   */
PBYTE pbOutBuf;      /* pointer to output buffer          */
USHORT cbOutBuf;     /* number of bytes in output buffer  */
PUSHORT pcbRead;     /* pointer to variable for bytes read */
ULONG ulTimeOut;     /* timeout value                     */
```

The **DosCallNmPipe** function opens a named pipe, writes to and reads from it, and closes it.

Parameters

pszName Points to the name of the pipe. The name is in the form **\pipe***name* for a local pipe and *****server***\pipe***name* for a remote pipe.

pbInBuf Points to the buffer containing the data that is written to the pipe.

cbInBuf Specifies the size (in bytes) of the input buffer.

pbOutBuf Points to the output buffer that receives the data read from the pipe.

cbOutBuf Specifies the size (in bytes) of the output buffer.

pcbRead Points to the variable that receives the number of bytes read from the pipe.

ulTimeOut Specifies a value (in milliseconds) that is the amount of time MS OS/2 should wait for the pipe to become available.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_INTERRUPT
ERROR_INVALID_FUNCTION
ERROR_SEM_TIMEOUT

Comments

The **DosCallNmPipe** function is equivalent to calling **DosOpen**, **DosTransact-NmPipe**, and **DosClose**.

See Also

DosMakePipe, **DosTransactNmPipe**

DosCaseMap

```
USHORT DosCaseMap(usLength, pctryc, pchString)
USHORT usLength;            /* length of string to casemap          */
PCOUNTRYCODE pctryc;        /* pointer to structure for country code */
PCHAR pchString;            /* pointer to character string          */
```

The **DosCaseMap** function casemaps the characters in the given string. If necessary, the function replaces characters in the string with the correct case-mapped characters.

The **DosCaseMap** function uses the casemap information in the *country.sys* file to casemap the string.

The **DosCaseMap** function is a family API function.

Parameters

usLength Specifies the length of the given string.

pctryc Points to the **COUNTRYCODE** structure that contains the country code and the code-page identifier for the casemap operation. The **COUNTRYCODE** structure has the following form:

```
typedef struct _COUNTRYCODE {
    USHORT country;
    USHORT codepage;
} COUNTRYCODE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pchString Points to the character string to be casemapped.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_NLS_BAD_TYPE
ERROR_NLS_NO_COUNTRY_FILE
ERROR_NLS_NO_CTRY_CODE
ERROR_NLS_OPEN_FAILED
ERROR_NLS_TABLE_TRUNCATED
ERROR_NLS_TYPE_NOT_FOUND

Restrictions

In real mode, the following restriction applies to the **DosCaseMap** function:

- There is no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

See Also

DosGetCollate, DosGetCtryInfo, DosSetCp

DosChDir

```
USHORT DosChDir(pszDirPath, ulReserved)
PSZ pszDirPath;        /* directory path */
ULONG ulReserved;      /* must be zero   */
```

The **DosChDir** function changes the current directory to the specified directory. When a process changes the current directory, subsequent calls to file-system functions, such as the **DosOpen** function, use the new directory as the default directory. The default directory is used if no explicit path is given with a filename.

The **DosChDir** function is a family API function.

Parameters

pszDirPath Points to the null-terminated string that specifies the new directory path. The string must be a valid MS OS/2 directory path and must not be longer than 125 characters.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_NOT_DOS_DISK
ERROR_NOT_ENOUGH_MEMORY
ERROR_PATH_NOT_FOUND

Comments

This function applies only to the process that is changing the directory. It does not affect the current directories of other processes. When the process terminates, the previous default directory becomes the default directory again.

When a process starts, it inherits its current directory from the parent process.

Example

This example stores the current default drive and path, then calls the **DosChDir** function to change the default path to the root directory:

```
PSZ pszPath;
USHORT cbPath = 0, usDisk;
ULONG ulLogicalDrives;
SEL selPath;

DosQCurDisk(&usDisk, &ulLogicalDrives);       /* gets current drive  */
DosQCurDir(usDisk, NULL, &cbPath);            /* gets size of buffer */
DosAllocSeg(cbPath, &selPath, SEG_NONSHARED); /* allocates memory    */
pszPath = MAKEP(selPath, 0);              /* assigns it to a far pointer */
DosQCurDir(usDisk, pszPath, &cbPath); /* gets current directory      */
DosChDir("\\", 0L);                  /* changes to the root directory */
    .
    .
    .
DosChDir(pszPath, 0L);               /* restores the directory       */
```

See Also

DosMkDir, DosQCurDir, DosQCurDisk, DosRmDir, DosSelectDisk

DosChgFilePtr

USHORT DosChgFilePtr(*hf, lDistance, fMethod, pulNewPtr***)**
HFILE *hf*; /* file handle */
LONG *lDistance*; /* distance to move */
USHORT *fMethod*; /* method of moving */
PULONG *pulNewPtr*; /* new pointer location */

The **DosChgFilePtr** function moves the file pointer to a new position in the file. The file pointer is maintained by the system. It points to the next byte to be read from a file or to the next position in the file to receive a byte.

The **DosChgFilePtr** function is a family API function.

Parameters

hf Identifies the file. This handle must have been created previously by using the **DosOpen** function.

lDistance Specifies the number of bytes to move the file pointer in the file. If this value is positive, the pointer moves forward through the file. If the value is negative, the pointer moves backward.

fMethod Specifies where the move will start. This parameter must be one of the following values:

Value	Meaning
FILE_BEGIN	Start move at the beginning of the file.
FILE_CURRENT	Start move at the current location.
FILE_END	Start move at the end of the file.

pulNewPtr Points to the long variable that receives the new file-pointer location.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_FUNCTION
ERROR_INVALID_HANDLE

Comments

The system automatically advances the file pointer for each byte read or written; the pointer is at the beginning of the file when the file is opened.

Example

This example opens the file *abc* for read and write access, calls the **DosChgFilePtr** function to set the file pointer at the end of the file, writes the string "Hello World", and closes the file. The *ulFilePointer* variable contains the file's current length when the pointer is at the end of the file.

```
HFILE hf;
USHORT usAction, cbBytesWritten;
ULONG ulFilePointer;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, 0L);
DosChgFilePtr(hf,                         /* file handle          */
    0L,                                   /* distance to move     */
    FILE_END,                             /* type of movement     */
    &ulFilePointer);                      /* address of new position */
DosWrite(hf, "Hello World\r\n", 13, &cbBytesWritten);
DosClose(hf);
```

See Also

DosNewSize, DosOpen, DosRead, DosWrite

DosCLIAccess

USHORT DosCLIAccess(*VOID*)

The **DosCLIAccess** function requests an input/output (I/O) privilege for disabling and enabling interrupts. Assembly-language programs that use the **cli** and **sti** instructions in **IOPL** segments must use the **DosCLIAccess** function to receive permission to use these instructions.

The **DosCLIAccess** function is a family API function.

This function has no parameters.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

Assembly-language programs that use the **in** and **out** instructions to read from and write to I/O ports must use the **DosPortAccess** function to receive permission to use these instructions. The **DosPortAccess** function also grants permission to use the **cli** and **sti** instructions.

See Also

DosPortAccess

DosClose

USHORT DosClose (*hf*)
HFILE *hf*; /* file handle */

The **DosClose** function closes a specified file or pipe. **DosClose** causes the system to write the contents of all the file's internal buffers to the device—for example, to the disk—and to update all directory information.

The **DosClose** function is a family API function.

Parameters

hf Identifies the file to close. This handle must have been created previously by using the **DosOpen** function, the **DosDupHandle** function, or the **DosMakePipe** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_FILE_NOT_FOUND
ERROR_INVALID_HANDLE

Example

This example opens the file *abc*, reads from the file, and calls the **DosClose** function to close it:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesRead;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
DosRead(hf, abBuf, sizeof(abBuf), &cbBytesRead);
DosClose(hf);                          /* closes the file */
```

See Also

DosBufReset, DosDupHandle, DosFindClose, DosMakePipe, DosOpen, DosRead

DosCloseQueue

USHORT DosCloseQueue (*hqueue*)
HQUEUE *hqueue*; /* queue handle */

The **DosCloseQueue** function closes a queue. If the process calling **DosCloseQueue** owns the queue, the function removes any outstanding elements from the queue. If the process does not own the queue, the contents of the queue remain unchanged and the queue remains available to other processes that have it open.

Parameters

hqueue Identifies the queue to be closed. This queue must have been previously created or opened by using the **DosCreateQueue** or **DosOpenQueue** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_QUE_INVALID_HANDLE

Comments

After the owner closes the queue, any process that attempts to write to the queue will receive an error value.

Example

This example creates and opens a queue, then calls the **DosCloseQueue** function to close the queue:

```
HQUEUE hqueue;
DosCreateQueue(&hqueue, QUE_FIFO, "\\queues\\abc.que");
    .
    .
    .
DosCloseQueue(hqueue);
```

See Also

DosCreateQueue, DosOpenQueue, DosReadQueue, DosWriteQueue

DosCloseSem

USHORT DosCloseSem(*hsem***)**
HSEM *hsem*; /* semaphore handle */

The **DosCloseSem** function closes a specified system semaphore. If another process has the semaphore open, it remains open and can be used by that process, although the semaphore cannot be used by the process that closes it. This function deletes the semaphore only when the last process using the semaphore closes it.

Parameters

hsem Identifies the semaphore to be closed. This handle must have been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_HANDLE
ERROR_SEM_IS_SET

Comments

If a process does not close its semaphores before terminating, the system closes them.

Example

This example opens a previously created system semaphore, then calls the **DosCloseSem** function to close it:

```
HSEM hsem;                                   /* semaphore handle     */
DosOpenSem(&hsem, "\\sem\\abc");              /* opens the semaphore  */
    .
    .
    .
DosCloseSem(hsem);                           /* closes the semaphore */
```

See Also

DosCreateSem, DosOpenSem

ConnectNmPipe

HORT DosConnectNmPipe (*hp*)
PIPE *hp*; /* pipe handle */

The **DosConnectNmPipe** function waits for a client to open a named pipe.

Parameters *hp* Identifies the named pipe. This handle must have been created previously by using **DosMakeNmPipe**.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_BROKEN_PIPE
ERROR_INTERRUPT
ERROR_INVALID_FUNCTION
ERROR_PIPE_NOT_CONNECTED

Comments If the client end of a named pipe is open, the **DosConnectNmPipe** function returns immediately. If the client end of a named pipe is not open and the pipe was created with blocking, the **DosConnectNmPipe** function waits until a client opens the pipe. If the client end of a named pipe is not open and the pipe was created with no blocking, the **DosConnectNmPipe** function returns an error value immediately.

In nonblocking mode, multiple **DosConnectNmPipe** calls can be issued to poll the state of a named pipe. If a client has not opened the pipe, the first call to the **DosConnectNmPipe** function puts the named pipe into a listening state and returns immediately with an ERROR_PIPE_NOT_CONNECTED return value. Subsequent calls to the **DosConnectNmPipe** function also return this error value, until a client opens the named pipe.

If a named pipe was opened and closed by a client but has not been disconnected by the controlling process, the **DosConnectNmPipe** function returns ERROR_BROKEN_PIPE.

See Also **DosDisConnectNmPipe**, **DosMakeNmPipe**

■ DosCreateCSAlias

USHORT DosCreateCSAlias (*selDataSegment*, *pselCodeSegment*)
SEL *selDataSegment*; /* data-segment selector */
PSEL *pselCodeSegment*; /* pointer to code-segment selector */

The **DosCreateCSAlias** function creates an aliased code-segment selector for a specified memory segment. The aliased code-segment selector can be used to pass execution control to machine instructions in a data segment.

The **DosCreateCSAlias** function is a family API function.

Parameters

selDataSegment Specifies the data-segment selector that identifies the memory segment.

pselCodeSegment Points to the variable that receives the aliased code-segment selector.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_ACCESS_DENIED

Comments

Shared memory segments, segments in huge memory blocks, and global data segments from dynamic-link libraries cannot be used to create an aliased code segment.

If the process has copied valid machine instructions to the data segment, the aliased code-segment selector can be combined with a segment offset to pass execution control to the machine instructions. The instructions in the aliased code segment can be called from either privilege level 2 (input/output privilege) or privilege level 3.

The **DosFreeSeg** function frees the aliased code-segment selector. Freeing the data-segment selector does not affect the aliased code segment, or vice versa. The segment is not removed from memory until both selectors have been freed.

Restrictions

In real mode, the following restrictions apply to the **DosCreateCSAlias** function:

- The selector returned is the address of the code.
- Freeing either the aliased selector or the original selector immediately frees the block of memory.

See Also

DosAllocSeg, DosFreeSeg

DosCreateQueue

```
USHORT DosCreateQueue(phqueue, fQueueOrder, pszQueueName)
PHQUEUE phqueue;        /* pointer to variable for queue handle      */
USHORT fQueueOrder;     /* order in which elements are read-written */
PSZ pszQueueName;       /* pointer to queue name                     */
```

The **DosCreateQueue** function creates and opens a queue. The new queue is owned by the process that calls the function, but can be opened for use by other processes.

Parameters

phqueue Points to the variable that receives the queue handle.

fQueueOrder Specifies the order in which elements are read from and written to the queue. This parameter can be one of the following values:

Value	Meaning
QUE_FIFO	First-in/first-out queue. The first element put in the queue is the first element to be removed.
QUE_LIFO	Last-in/first-out queue. The last element put in the queue is the first element to be removed.
QUE_PRIORITY	Priority queue. The process that places the element in the queue specifies a priority. Elements with the highest priority are removed first.

pszQueueName Points to a null-terminated string. The string identifies the queue and must have the following form:

\queues*name*

The string name, *name*, must have the same format as an MS OS/2 filename and must be unique.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_QUE_DUPLICATE
ERROR_QUE_INVALID_NAME
ERROR_QUE_INVALID_PRIORITY
ERROR_QUE_NO_MEMORY

Comments

The process that creates a queue owns that queue. The owning process can write elements to and read elements from the queue at any time, since **DosCreateQueue** automatically opens the queue for the owning process. Other processes may open the queue by using the **DosOpenQueue** function and write elements to it by using the **DosWriteQueue** function, but they cannot read elements from the queue. Any thread belonging to the process that owns a queue can read from or write to the queue.

If any process has a queue open when the owner closes it, subsequent requests to write to the queue return an error value.

See Also

DosCloseQueue, DosOpenQueue

DosCreateSem

```
USHORT DosCreateSem(fNoExclusive, phssm, pszSemName)
USHORT fNoExclusive;    /* exclusive/nonexclusive ownership flag    */
PHSYSSEM phssm;         /* pointer to variable for semaphore handle */
PSZ pszSemName;         /* pointer to semaphore name                */
```

The **DosCreateSem** function creates a system semaphore and copies the semaphore handle to a variable. A process can use a system semaphore to indicate to another process a change in the status of a shared resource.

Parameters

fNoExclusive Specifies ownership of the semaphore. If this parameter is CSEM_PRIVATE, the process receives exclusive ownership. If this parameter is CSEM_PUBLIC, the process does not receive exclusive ownership.

phssm Points to the variable that receives the semaphore handle.

pszSemName Points to a null-terminated string that identifies the semaphore. The string must have the following form:

\sem*name*

The string name, *name*, must have the same format as an MS OS/2 filename and must be unique.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ALREADY_EXISTS
ERROR_INVALID_NAME
ERROR_INVALID_PARAMETER
ERROR_TOO_MANY_SEMAPHORES

Comments

The process that creates the system semaphore owns it. Other processes can open the semaphore by using the **DosOpenSem** function, then wait for a change in the status of the semaphore by using the **DosSemWait** or **DosMuxSemWait** function. The owning process can change the status of the semaphore by using the **DosSemSet** or **DosSemClear** functions.

The process calling the **DosCreateSem** function receives exclusive ownership of a system semaphore, unless otherwise specified. Exclusive ownership prevents other processes from setting or clearing the semaphore while the owning process has it open. Other processes may open the semaphore and wait for it to change status, but they cannot change its status.

Example

This example calls **DosCreateSem** to create a system semaphore, then calls **DosSemSet** to set it and **DosSemClear** to clear it:

```
HSYSSEM hssm;                  /* handle to semaphore */
DosCreateSem(CSEM_PRIVATE,     /* specifies ownership */
    &hssm,                     /* address of handle   */
    "\\sem\\abc.sem");         /* name of semaphore   */
DosSemSet(hssm);               /* sets the semaphore  */
    .
    .
    .
DosSemClear(hssm);             /* clears the semaphore */
```

See Also

DosCloseSem, DosOpenSem, DosSemClear, DosSemRequest, DosSemSet, DosSemSetWait, DosSemWait

DosCreateThread

```
USHORT DosCreateThread(pfnFunction, ptidThread, pbThrdStack)
PFNTHREAD pfnFunction(VOID);   /* pointer to address of function        */
PTID ptidThread;               /* pointer to variable for thread identifier */
PBYTE pbThrdStack;             /* pointer to thread stack               */
```

The **DosCreateThread** function creates a new thread.

Parameters

pfnFunction Points to a program-supplied function and represents the starting address of the thread. For a full description, see the following "Comments" section.

ptidThread Points to the variable that receives the thread identifier.

pbThrdStack Points to the address of the new thread's stack.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_NO_PROC_SLOTS
ERROR_NOT_ENOUGH_MEMORY

Comments

When a thread is created, the system makes a far call to the application-supplied function whose address is specified by the *pfnFunction* parameter. This function may include local variables and may call other functions, as long as the thread's stack has sufficient space. (The stack can be allocated by using the **DosAllocSeg** function or by using a global array.) The address specified by the *pbThrdStack* parameter should be the address of the last word in the stack, not the first, since the stack grows down in memory. The thread terminates when the function returns or calls the **DosExit** function.

The *pfnFunction* parameter points to a function that is supplied by the program. This function should have the following form:

```
VOID FAR FuncName(VOID)
{
}
```

Since the system passes no arguments, no parameters are defined.

DosCreateThread can create up to 255 threads per process. A new thread inherits all files and resources owned by the parent process. Any thread in a process can open a file, device, pipe, queue, or system semaphore. Other threads may use the corresponding handles to access the given item.

Note that high-level-languages, run-time libraries, and stack checking may severely limit or eliminate the ability to call the **DosCreateThread** function directly from a high-level-language program. For more information, consult the documentation that came with your language product.

Before calling the **DosCreateThread** function, either set the **es** register to zero or assign to it a selector that will remain valid for the duration of the new thread. If you fail to set the **es** register to one of these values, the thread may unexpectedly terminate as a result of a general protection fault. For more information, see the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

Example

This example sets aside a 512-byte buffer to be used as stack space for any threads that are created. The first stack is set at the end of the array. The thread is created by calling the **DosCreateThread** function. The thread terminates by calling the **DosExit** function.

```
VOID FAR Thread1();
BYTE abStackArea[512];
    .
    .
    .
    PVOID pStack1 = abStackArea + 512;   /* 512-byte stack          */
    TID tidThread1;

    DosCreateThread(Thread1,             /* name of thread function */
        &tidThread1,                     /* address of thread ID    */
        pStack1);                        /* thread's stack          */
    .
    .
    .
    DosExit(EXIT_PROCESS, 0);
}

VOID FAR Thread1() {
    .
    .
    .
    DosExit(EXIT_THREAD, 0);
}
```

See Also

DosExit, DosResumeThread, DosSuspendThread

■ DosCwait

```
USHORT DosCwait(fScope, fWait, prescResults, ppidProcess, pidWaitProcess)
USHORT fScope;                 /* flag scope                              */
USHORT fWait;                  /* wait/no-wait flag                       */
PRESULTCODES prescResults;     /* pointer to structure receiving result codes */
PPID ppidProcess;              /* pointer to variable for process identifier  */
PID pidWaitProcess;            /* process identifier of process to wait for   */
```

The **DosCwait** function waits for a child process to terminate, then retrieves the result codes from that process. The function copies the process identifier of the terminated process to the variable pointed to by the *ppidProcess* parameter and copies a termination code to the structure pointed to by the *prescResults* parameter.

Parameters

fScope Specifies how many processes to wait for. If the value of this parameter is DCWA_PROCESS, the thread waits until the specified process ends. If it is DCWA_PROCESSTREE, the thread waits until the specified process and all its child processes end.

fWait Specifies whether or not to wait for child processes. If this parameter is DCWW_WAIT, the thread waits while child processes are running. If it is DCWW_NOWAIT, the thread does not wait. This option is used to retrieve the result codes of a child process that has already ended.

prescResults Points to the **RESULTCODES** structure that receives the termination code and result code for the child process's termination. The **RESULTCODES** structure has the following form:

```
typedef struct _RESULTCODES {
    USHORT codeTerminate;
    USHORT codeResult;
} RESULTCODES;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

ppidProcess Points to the variable that receives the process identifier of the ending process.

pidWaitProcess Specifies which process to wait for. If this parameter is a process identifier, the thread waits for that process to end. If it is zero, the thread waits until any child process ends.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_CHILD_NOT_COMPLETE
ERROR_INVALID_PROCID
ERROR_WAIT_NO_CHILDREN

Comments

The **DosCwait** function may wait for a child process and any processes started by the child process to end before it returns, but it will not report the status of the processes that were started by the child process.

When the function is waiting for more then one child process, the *ppidProcess* variable is used to determine which child process has terminated.

Do not call the **DosCwait** function before starting a child process. When this happens, the process calling **DosCwait** waits indefinitely, since a child process cannot start asynchronously.

Example

This example runs the *cmd.exe* program as a child process, then calls the **DosCwait** function to wait until *cmd.exe* terminates:

```
CHAR achFailName[128];
RESULTCODES rescResults;
PID pidProcess;
DosExecPgm(achFailName, sizeof(achFailName),
    EXEC_ASYNC, "cmd ", 0, &rescResults, "cmd.exe");
    .
    .
    .
DosCwait(DCWA_PROCESS,              /* execution flag              */
    DCWW_WAIT,                      /* wait option                 */
    &rescResults,                   /* address for result codes    */
    &pidProcess,                    /* address of process identifier */
    rescResults.codeTerminate);     /* process to wait for         */
```

See Also

DosExecPgm, DosExit, DosKillProcess

DosDelete

USHORT DosDelete(*pszFileName, ulReserved***)**
PSZ *pszFileName*; /* pointer to string specifying pathname */
ULONG *ulReserved*; /* must be zero */

The **DosDelete** function deletes a file.

The **DosDelete** function is a family API function.

Parameters

pszFileName Points to a null-terminated string that specifies the file to be deleted. This string must be a valid MS OS/2 filename and must not contain wildcard characters.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_FILE_NOT_FOUND
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION

Comments

Read-only files cannot be deleted by using the **DosDelete** function. The **DosSetFileMode** function can be used to change a file's read-only attributes, making it possible to delete that file.

The **DosDelete** function cannot delete directories; use the **DosRmDir** function to delete directories.

Example

This example calls the **DosDelete** function to delete the file *abc*, and displays a message reporting success or failure:

```
USHORT usError;
usError = DosDelete("abc", 0L);
if (usError)
    VioWrtTTY("abc not deleted\r\n", 21, 0);
else
    VioWrtTTY("abc deleted\r\n", 17, 0);
```

See Also

DosRmDir, DosSetFileMode

DosDevConfig

USHORT DosDevConfig(*pvDevInfo, usItem, usReserved***)**
PVOID *pvDevInfo*; /* pointer to variable for device information */
USHORT *usItem*; /* item number */
USHORT *usReserved*; /* must be zero */

The **DosDevConfig** function retrieves information about attached devices.

The **DosDevConfig** function is a family API function.

Parameters

pvDevInfo Points to the variable that receives device information. The type of information received depends on the value of the *usItem* parameter.

usItem Specifies what device information to retrieve. This parameter can be one of the following values:

Value	Meaning
DEVINFO_ADAPTER	The *pvDevInfo* parameter points to a **BYTE** variable that is set to FALSE if the primary display adapter is a monochrome/printer display adapter type, or to TRUE for other display adapters.
DEVINFO_COPROCESSOR	The *pvDevInfo* parameter points to a **BYTE** variable that is set to TRUE if a math coprocessor is present.
DEVINFO_FLOPPY	The *pvDevInfo* parameter points to a **USHORT** variable that receives the number of removeable-disk drives that are installed.
DEVINFO_MODEL	The *pvDevInfo* parameter points to a **BYTE** variable that receives the PC model type.
DEVINFO_PRINTER	The *pvDevInfo* parameter points to a **USHORT** variable that receives the number of printers that are attached.
DEVINFO_RS232	The *pvDevInfo* parameter points to a **USHORT** variable that receives the number of RS232 cards that are attached.
DEVINFO_SUBMODEL	The *pvDevInfo* parameter points to a **BYTE** variable that receives the PC submodel type.

usReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_PARAMETER

Example

This example calls the **DosDevConfig** function to determine if a math coprocessor is present:

```
BYTE bDevInfo;
DosDevConfig(&bDevInfo,      /* address of variable for device info. */
    DEVINFO_COPROCESSOR,     /* information requested                */
    0);                      /* reserved                             */
if (bDevInfo)
    VioWrtTTY("Math coprocessor present\r\n", 26, 0);
else
    VioWrtTTY("Math coprocessor not present\r\n", 30, 0);
```

See Also

DosDevIOCtl, VioGetConfig

■ DosDevIOCtl

```
USHORT DosDevIOCtl(pvData, pvParms, usFunction, usCategory, hDevice)
PVOID pvData;          /* pointer to buffer for data area          */
PVOID pvParms;         /* pointer to buffer for command arguments  */
USHORT usFunction;     /* device function                          */
USHORT usCategory;     /* device category                          */
HFILE hDevice;         /* device handle                            */
```

The **DosDevIOCtl** function passes device-control functions to the device specified by the *hDevice* parameter.

The **DosDevIOCtl** function is a family API function.

Parameters

pvData Points to a buffer that receives data from the given control function. Some control functions may also read data from the buffer as part of their processing.

pvParms Points to a buffer that contains any data required for the given control function. Some control functions may copy data to the buffer as part of their processing.

usFunction Specifies the device-control function. This parameter can be any one of the device-control function codes described in Chapter 3, "Input-and-Output Control Functions."

usCategory Specifies the device categories. This parameter can be any one of the device categories described in Chapter 3, "Input-and-Output Control Functions."

hDevice Identifies the device that receives the device-control function. This handle must have been created previously by using the **DosOpen** function or it must be a standard (open) device handle.

Return Value

In addition to the system error values, the **DosDevIOCtl** function returns device driver return-value information. Return values in the range 0xFF00 through 0xFFFF are user-dependent error values. Return values in the range 0xFE00 through 0xFEFF are device-driver-dependent error values.

The error value may be one of the following:

ERROR_BAD_DRIVER_LEVEL
ERROR_INVALID_CATEGORY
ERROR_INVALID_DRIVE
ERROR_INVALID_FUNCTION
ERROR_INVALID_HANDLE
ERROR_PROTECTION_VIOLATION

Restrictions

In real mode, the following restrictions apply to the **DosDevIOCtl** function:

- Some control functions in categories 1, 5, and 8 can be used with MS-DOS 3.*x*, but not with MS-DOS 2.*x*.
- Categories 2, 3, 4, 6, 7, 10, and 11 cannot be used.

Example

This example calls the **DosDevIOCtl** function to change the typamatic rate of the keyboard. Before you can use the **DosDevIOCtl** function to access the keyboard you must open the keyboard device and set the focus.

```
USHORT usParameters[2];
HKBD hkbd;
usParameters[0] = 500;              /* delay in milliseconds     */
usParameters[1] = 60;               /* characters per second     */
KbdOpen(&hkbd);                     /* opens the keyboard        */
KbdGetFocus(0, hkbd);               /* gets the focus            */
DosDevIOCtl(0L,                     /* data area                 */
    (PCHAR) usParameters,           /* command arguments         */
    0x54,                           /* function code             */
    4,                              /* device category           */
    hkbd);                          /* handle to device keyboard */
```

See Also

DosOpen, KbdGetFocus, KbdOpen

DosDisConnectNmPipe

USHORT DosDisConnectNmPipe(*hp***)**
HPIPE *hp*; /* pipe handle */

The **DosDisConnectNmPipe** function closes a client's handle of a named pipe.

Parameters

hp Identifies the named pipe. This handle must have been created previously by using the **DosMakeNmPipe** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_INVALID_FUNCTION

Comments

If the client end of a named pipe is open, the **DosDisConnectNmPipe** function forces that end of the named pipe closed. The client will receive an error value the next time it attempts to access the pipe. Closing the client end of a named pipe may discard data in the pipe before the client reads the data.

A client that is forced off a pipe by a call to **DosDisConnectNmPipe** must still close its end of the pipe by using the **DosClose** function.

See Also

DosClose, DosConnectNmPipe, DosMakeNmPipe

DosDupHandle

USHORT DosDupHandle(*hfOld*, *phfNew***)**
HFILE *hfOld*; /* handle of existing file */
PHFILE *phfNew*; /* pointer to variable containing new file handle */

The **DosDupHandle** function duplicates a file handle. The new handle has the same handle-specific information as the existing handle, such as its file-pointer position and access method. The original handle and the duplicate are interchangeable, since most changes to one affect the other. For example, moving the file pointer for the original handle moves the pointer for the new handle. Closing the original handle by using the **DosClose** function does not close the duplicate handle, however, and closing the duplicate does not close the original. A file is not closed until its last handle is closed.

The **DosDupHandle** function is a family API function.

Parameters

hfOld Identifies the file handle to duplicate. This handle must have been created previously by using the **DosOpen** function. The **DosDupHandle** function closes the file before duplicating its handle.

phfNew Points to the variable that contains the new file handle. If this parameter is 0xFFFF, the **DosDupHandle** function creates a new handle and copies it to the variable pointed to by the *phfNew* parameter. Any specified value other than 0xFFFF is used as the handle.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_HANDLE
ERROR_INVALID_TARGET_HANDLE
ERROR_TOO_MANY_OPEN_FILES

Comments

You can change the inheritance, fail-on-error, and write-through flags for the duplicate file handle by using the **DosSetFHandState** function.

Example

This example calls the **DosDupHandle** function to duplicate the standard output handle, and then writes "Hello World" to the new handle:

```
HFILE hfNew;
USHORT, cbBytesWritten;
hfNew = 0xFFFF;                    /* create new handle        */
DosDupHandle(1, &hfNew);           /* duplicate standard output */
DosWrite(hfNew, "Hello World\r\n", 13, &cbBytesWritten);
```

See Also

DosChgFilePtr, DosClose, DosExecPgm, DosMakePipe, DosRead, DosSetFHandState, DosWrite

DosEnterCritSec

VOID DosEnterCritSec(*VOID*)

The **DosEnterCritSec** function suspends every thread in the current process, except for the calling thread. Suspended threads will not execute until the current thread calls the **DosExitCritSec** function.

This function has no parameters.

Return Value

This function does not return a value.

Comments

The signal handler (if installed) is not suspended when the **DosEnterCritSec** function is called. If a signal occurs, the processing done by the signal handler must not interfere with the processing done by the thread calling the **DosEnterCritSec** function.

See Also

DosCreateThread, DosExitCritSec, DosHoldSignal, DosSetSigHandler

■ DosErrClass

```
USHORT DosErrClass(usErrorCode, pusClass, pfsAction, pusLocus)
USHORT usErrorCode;     /* error value for analysis                 */
PUSHORT pusClass;       /* pointer to variable for error classification */
PUSHORT pfsAction;      /* pointer to variable for action           */
PUSHORT pusLocus;       /* pointer to variable for error origin     */
```

The **DosErrClass** function retrieves a classification of an error value and a recommended action.

The **DosErrClass** function is a family API function.

Parameters

usErrorCode Specifies the error value returned by an MS OS/2 function.

pusClass Points to the variable that receives the classification of the error value. This parameter can be one of the following values:

Value	Meaning
ERRCLASS_ALREADY	Action already taken.
ERRCLASS_APPERR	An application error has probably occurred.
ERRCLASS_AUTH	Authorization has failed.
ERRCLASS_BADFMT	Bad format for call data.
ERRCLASS_CANT	Cannot perform requested action.
ERRCLASS_HRDFAIL	A device-hardware failure has occurred.
ERRCLASS_INTRN	An internal error has occurred.
ERRCLASS_LOCKED	Resource or data is locked.
ERRCLASS_MEDIA	Incorrect media; a CRC error has occurred.
ERRCLASS_NOTFND	The item was not located.
ERRCLASS_OUTRES	Out of resources.
ERRCLASS_SYSFAIL	A system failure has occurred.
ERRCLASS_TEMPSIT	This is a temporary situation.
ERRCLASS_TIME	A time-out has occurred.
ERRCLASS_UNK	The error is unclassified.

pfsAction Points to the variable that receives the recommended action for the specific error. This parameter can be one of the following values:

Value	Meaning
ERRACT_ABORT	Terminate in an orderly manner.
ERRACT_DLYRET	Delay and retry.
ERRACT_IGNORE	Ignore the error.
ERRACT_INTRET	Retry after user intervention.
ERRACT_PANIC	Terminate immediately.
ERRACT_RETRY	Retry immediately.
ERRACT_USER	Bad user input; get new values.

pusLocus Points to the variable that receives the error's origin in the system. This parameter can be one of the following values:

Value	Meaning
ERRLOC_DISK	The error occurred in a random-access device, such as a disk drive.
ERRLOC_MEM	This is a memory-parameter error.
ERRLOC_NET	This is a network error.
ERRLOC_SERDEV	This is a serial-device error.
ERRLOC_UNK	The origin of the error is unknown.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The ERRACT_, ERRCLASS_, and ERRLOC_ constants are defined in the *bseerr.h* file.

Example

This example calls the **DosQFileMode** function to determine the status of the file *a:\abc.exe*. If **DosQFileMode** returns an error, the **DosErrClass** function is called to determine the class of the error. The process terminates if the error is a device-hardware failure—for example, if a drive door is open or a specified disk drive is nonexistent.

```
USHORT usAttribute, usError, usClass, fsAction, usLocus;
usError = DosQFileMode("a:\\abc.ext", &usAttribute, 0L);
if (usError) {
    DosErrClass(usError,                /* error number           */
        &usClass,                       /* error classification   */
        &fsAction,                      /* recommended action     */
        &usLocus);                      /* error origin           */
    if (usClass == ERRCLASS_HRDFAIL)   /* device-hardware failure */
        DosExit(1, EXIT_PROCESS);       /* exits application      */
```

See Also

DosError, DosExit, DosQFileMode

■ DosError

USHORT DosError(*fEnable***)**
USHORT *fEnable*; /* enable/disable error handling */

The **DosError** function enables or disables hard-error and exception processing for a process. By default, the system displays a message and prompts for user input when a hard error or exception occurs. A hard error is typically an error that cannot be resolved by software—for example, when the drive door is opened while a removeable disk is being read.

The **DosError** function disables the default processing by forgoing the displayed message and directing any function that encounters a hard error or exception to return an appropriate error value. The process must determine the appropriate action by referring to the error value.

The **DosError** function is a family API function.

Parameters

fEnable Specifies whether to disable or enable processing. This parameter can be one of the following values:

Value	Meaning
EXCEPTION_DISABLE	Disable exception processing.
EXCEPTION_ENABLE	Enable exception processing.
HARDERROR_DISABLE	Disable hard-error processing.
HARDERROR_ENABLE	Enable hard-error processing.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_DATA

Comments

By default, the system terminates any process in which an exception occurs. Although the **DosError** function can disable the message when an exception occurs, it cannot disable the termination of the process. To prevent a process from being terminated, use the **DosSetVec** function to trap the exception and carry out process-specific exception processing.

Restrictions

In real mode, the following restriction applies to the **DosError** function:

- If the *fEnable* parameter is HARDERROR_DISABLE, all subsequent **int 24h** requests fail until a call is made to the **DosError** function with *fEnable* set to HARDERROR_ENABLE.

Example

This example calls the **DosError** function to turn off hard-error processing, then calls the **DosErrClass** function to process any error that is received:

```
USHORT usAttribute, usError, usClass, fsAction, usLocus;
DosError(HARDERROR_DISABLE);        /* turn off hard-error processing */
usError = DosQFileMode("a:\\abc.ext", &usAttribute, 0L);
if (usError) {
    DosErrClass(usError, &usClass, &fsAction, &usLocus);
    if (usClass == ERRCLASS_HRDFAIL)
        DosExit(1, EXIT_PROCESS);
```

See Also

DosErrClass, DosSetFHandState

DosExecPgm

```
USHORT DosExecPgm(pchFailName, cbFailName, fExecFlags, pszArgs, pszEnv, prescResults, pszPgmName)
PCHAR pchFailName;            /* pointer to buffer for failed filename      */
SHORT cbFailName;             /* size of failed filename buffer             */
USHORT fExecFlags;            /* synchronous/trace flags                    */
PSZ pszArgs;                  /* pointer to argument strings                */
PSZ pszEnv;                   /* pointer to environment strings             */
PRESULTCODES prescResults;    /* pointer to structure receiving result codes */
PSZ pszPgmName;               /* pointer to program name to execute         */
```

The **DosExecPgm** function loads and starts a child process.

The **DosExecPgm** function is a family API function.

Parameters

pchFailName Points to the buffer that receives the name of the object (such as a dynamic-link module). The **DosExecPgm** function copies a name to this buffer if it cannot load and start the specified program.

cbFailName Specifies the length (in bytes) of the buffer pointed to by the *pchFailName* parameter.

fExecFlags Specifies how a given program should be run. This parameter can be one of the following values:

Value	Meaning
EXEC_ASYNC	Execute asynchronously to the parent process. The **DosExecPgm** function copies the process identifier of the child process to the **codeTerminate** field of the structure pointed to by the *prescResults* parameter.
EXEC_ASYNCRESULT	Execute asynchronously to the parent process. Before returning, the **DosExecPgm** function copies the process identifier of the child process to the **codeTerminate** field of the structure pointed to by the *prescResults* parameter. When the child process ends, the system saves the termination and result codes in memory it reserves for these codes. This memory remains allocated until the parent process calls the **DosCwait** function to retrieve the information.
EXEC_SYNC	Execute synchronously to the parent process. When the child process ends, the **DosExecPgm** function copies its termination and result codes to the structure pointed to by the *prescResults* parameter.
EXEC_BACKGROUND	Execute asynchronously to the parent process and detach from the screen group of the parent process. The detached process executes in the background. If a process terminates the parent process—for example, by using the **DosKillProcess** function—the child process continues to run. The child process should not require screen output (other than through the **VioPopUp** function). The child process also should not call **Vio**, **Kbd**, or **Mou** functions.
EXEC_TRACE	Execute under conditions for tracing. The parent process debugs the child process.

pszArgs Points to a set of null-terminated argument strings that represent the program's command parameters. The argument strings are copied to the process's environment segment. The string can have any format but must end with two null characters. A typical format is the program name, a null character, the program parameters (separated by spaces), and two null characters.

If this parameter is zero, no argument strings are passed to the child process.

pszEnv Points to a set of null-terminated environment strings that represent environment variables and their current values. The environment strings are copied to the process's environment segment. These strings represent environment variables and their current values. An environment string has the following form:

variable=value

Two or more strings can be concatenated to pass multiple environment strings to the child process. The last environment string must end with two null characters.

If this parameter is zero, the child process inherits the unchanged environment of the parent process.

prescResults Points to the **RESULTCODES** structure that receives the termination and result codes of the child process. The **RESULTCODES** structure has the following form:

```
typedef struct _RESULTCODES {
    USHORT codeTerminate;
    USHORT codeResult;
} RESULTCODES;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pszPgmName Points to a null-terminated string that specifies the process to load and start. The string must be a valid MS OS/2 filename and include the filename extension. The string must specify an executable file.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_AUTODATASEG_EXCEEDS_64k
ERROR_BAD_ENVIRONMENT
ERROR_BAD_FORMAT
ERROR_DRIVE_LOCKED
ERROR_DYNLINK_FROM_INVALID_RING
ERROR_EXE_MARKED_INVALID
ERROR_FILE_NOT_FOUND
ERROR_INTERRUPT
ERROR_INVALID_DATA
ERROR_INVALID_EXE_SIGNATURE
ERROR_INVALID_FUNCTION
ERROR_INVALID_MINALLOCSIZE
ERROR_INVALID_MODULETYPE
ERROR_INVALID_ORDINAL
ERROR_INVALID_SEGDPL
ERROR_INVALID_SEGMENT_NUMBER
ERROR_INVALID_STACKSEG
ERROR_INVALID_STARTING_CODESEG
ERROR_ITERATED_DATA_EXCEEDS_64K
ERROR_LOCK_VIOLATION
ERROR_NO_PROC_SLOTS
ERROR_NOT_DOS_DISK
ERROR_NOT_ENOUGH_MEMORY
ERROR_PATH_NOT_FOUND
ERROR_PROC_NOT_FOUND

ERROR_RELOC_CHAIN_XEEDS_SEGLIM
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION
ERROR_TOO_MANY_OPEN_FILES

Comments

If the filename is a complete pathname (a drive name, path, and filename), the **DosExecPgm** function loads the program from the specified location. If only a filename is given and that filename is not found in the current directory, the **DosExecPgm** function searches each directory specified in the parent process's **PATH** environment variable for the given file.

The child process receives a discrete address space—that is, it receives its own local descriptor table. This means that the parent process and the child process cannot access each other's data. To pass data between processes, the parent process typically opens a pipe by using the **DosMakePipe** function before starting the child process, then lets the child process access one end of the pipe.

The environment segment of the child process consists of the environment strings (at offset zero), the program filename, and the argument strings. The system passes the offset to the argument strings in the **bx** register and the environment segment's selector in the **ax** register. These values can also be retrieved by using the **DosGetEnv** function.

When the child process starts, it inherits all pipe handles and all open file handles from the parent process. (File handles that are opened with the *fsOpenMode* parameter of the **DosOpen** function set to OPEN_FLAGS_NOINHERIT are not inherited by the child process—for more information, see the **DosOpen** function.) The child process can use these handles immediately, without opening or preparing them in any way. This gives the parent process control over the files associated with the standard input, output, and error file handles. For example, the parent process can redirect the standard output from the screen to a file by opening the file and duplicating its handle as the standard output handle (0x0001). If the child process then writes to the standard output, the data goes to the file, not to the screen.

Restrictions

In real mode, the following restrictions apply to the **DosExecPgm** function:

- The only value allowed for the *fExecFlags* parameter is EXEC_SYNC.
- The buffer pointed to by the *pchFailName* parameter is filled with blanks, even if the function fails.
- The **codeResult** field of the **RESULTCODES** structure receives the exit code from either the **DosExit** function or the MS-DOS **int 21h, 4cH** system call, whichever is used to terminate the program.

Example

This example calls the **DosExecPgm** function to execute the program *abc.exe*. The program executes as a child process asynchronously with the parent program.

```
CHAR achFailName[128];
RESULTCODES rescResults;
DosExecPgm(achFailName,                 /* object-name buffer */
    sizeof(achFailName),                /* length of buffer   */
    EXEC_ASYNCH,                        /* async flag         */
    "abc = 0\0",                        /* argument string    */
    0,                                  /* environment string */
    &rescResults,                       /* address of result  */
    "abc.exe");                         /* name of program    */
```

See Also

DosCreateThread, DosCwait, DosExit, DosGetEnv, DosKillProcess, DosOpen

DosExit

VOID DosExit(*fTerminate*, *usExitCode* **)**
USHORT *fTerminate*; /* terminate current/all threads */
USHORT *usExitCode*; /* result code for parent process */

The **DosExit** function ends a thread or a process and all its threads.

The **DosExit** function is a family API function.

Parameters

fTerminate Specifies whether to terminate the current thread or the process and all its threads. If this parameter is EXIT_THREAD, only the current thread ends. If it is EXIT_PROCESS, all threads in the process end.

usExitCode Specifies the program's exit code.

Return Value

This function does not return a value.

Comments

If the *fTerminate* parameter is EXIT_THREAD, the function ends the current thread. If the current thread is the last one in the process, the process also ends. If the *fTerminate* parameter is EXIT_PROCESS, the **DosExit** function terminates all threads in the process and creates a final temporary thread. The temporary thread executes any functions given in the list created by the **DosExitList** function. When this last thread ends, the system frees any resources used by the process. The exit code specified by the last call to the **DosExit** function is supplied to the parent process by using the **DosCwait** function.

Restrictions

In real mode, the following restriction applies to the **DosExit** function:

- The function always exits from the current program, since there are no threads in the real-mode environment.

Example

This example creates a thread, referred to as thread 2. This example shows two ways of stopping thread 2: by stopping all threads in the process and by stopping thread 2 specifically. Thread 1, the main process, exits and ends all threads by calling the **DosExit** function with the first parameter set to EXIT_PROCESS. Thread 2, the thread created with the call to **DosCreateThread**, ends only itself, by calling **DosExit** with the first parameter set to EXIT_THREAD.

```
BYTE bStackArea[2048];

main() {
    .
    .
    .
    PVOID pStack2 = bStackArea + 512;
    TID tidThread2;
    DosCreateThread(Thread2, &tidThread2, pStack2);
    .
    .
    .
    DosExit(EXIT_PROCESS,          /* exit process                         */
        0);                        /* return value                         */
}
VOID FAR Thread2() {
    .
    .
    .
    DosExit(EXIT_THREAD,           /* exit thread, process continues */
        0);                        /* return value                         */
}
```

See Also

DosCwait, DosExecPgm, DosExitList

DosExitCritSec

VOID DosExitCritSec(*VOID* **)**

The **DosExitCritSec** function restores execution of all threads in the process that were suspended by the **DosEnterCritSec** function.

This function has no parameters.

Return Value This function does not return a value.

See Also **DosCreateThread, DosEnterCritSec**

DosExitList

USHORT DosExitList(*fFnCode, pfnFunction* **)**
USHORT *fFnCode*; /* function code */
PFNEXITLIST *pfnFunction(USHORT)*; /* pointer to address of function */

The **DosExitList** function specifies a function that is executed when the current process ends. This "termination function" may define additional termination functions. The **DosExitList** function may be called one or more times: each call adds or subtracts a function from an internal list that is maintained by the system. When the current process terminates, MS OS/2 transfers control to each function on the list.

Parameters

fFnCode Specifies whether a function's address is added to or removed from the list. This parameter can be one of the following values:

Value	Meaning
EXLST_ADD	Add function to termination list.
EXLST_EXIT	Termination processing complete; call the next function on termination list.
EXLST_REMOVE	Remove function from termination list.

pfnFunction Points to the termination function to be added to the list. For a full description, see the following "Comments" section.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_DATA
ERROR_NOT_ENOUGH_MEMORY

Comments Dynamic-link-library modules often use the **DosExitList** function; this function allows library modules to free resources or clear flags and semaphores if the client process terminates without notifying them.

The termination function has one parameter and no return value. The function should have the following form:

```
PFNEXITLIST FuncName(usTermCode)
USHORT usTermCode;
{
    .
    .
    .
    DosExitList(EXLST_EXIT, 0);
}
```

The *usTermCode* parameter of the termination function specifies the reason the process ended. This parameter is one of the following values:

Value	Meaning
TC_EXIT	Normal exit
TC_HARDERROR	Hard-error abort
TC_KILLPROCESS	Unintercepted **DosKillProcess**
TC_TRAP	Trap operation

Before transferring control to the termination function, MS OS/2 resets the stack to its initial value. MS OS/2 then passes control to the function by using a **jmp** instruction. The termination function should carry out its tasks and then call the **DosExitList** function with the *fFnCode* parameter set to EXLST_EXIT. This parameter setting directs the system to call the next function on the termination list. When all functions on the list have been called, the process ends.

Termination functions should be as short and fail-safe as possible. When the termination functions are executed, all threads except for the one executing the **DosExitList** function have been destroyed. A termination function must call the **DosExitList** function to end; otherwise, the process "hangs," since MS OS/2 cannot terminate it.

A termination function can call most MS OS/2 system functions; however, it must not call the **DosCreateThread** or **DosExecPgm** function.

Example

This example calls the **DosExitList** function, which then adds the locally defined function CleanUp to the list of routines to be called when the process terminates. The CleanUp function displays a message that it is cleaning up, then calls **DosExitList**, reporting that it has finished and that the next function on the termination list can be called.

```
    DosExitList(EXLST_ADD,                /* adds address to the list */
        CleanUp);                         /* function address         */
    .
    .
    .
    DosExit(EXIT_PROCESS, 0);
}

VOID PASCAL FAR CleanUp(usTermCode)
USHORT usTermCode;
{
    VioWrtTTY("Cleaning up...\r\n", 16, 0);
    .
    .
    .
    DosExitList(EXLST_EXIT,               /* termination complete     */
        0L);
}
```

See Also

DosCreateThread, DosExecPgm, DosExit

DosFileLocks

```
USHORT DosFileLocks(hf, pfUnlock, pfLock)
HFILE hf;                /* file handle                    */
PFILELOCK pfUnLock;      /* pointer to range to be unlocked */
PFILELOCK pfLock;        /* pointer to range to be locked   */
```

The **DosFileLocks** function unlocks and/or locks a region in an open file. Locking a region prevents other processes from accessing the locked region.

The **DosFileLocks** function is a family API function.

Parameters

hf Identifies the file handle. This handle must have been created previously by using the **DosOpen** function.

pfUnLock Points to the **FILELOCK** structure that specifies the starting position in the file and the number of bytes of the file to unlock. This parameter is ignored if NULL is specified instead of a structure address. The **FILELOCK** structure has the following form:

```
typedef struct _FILELOCK {
    LONG lOffset;
    LONG lRange;
} FILELOCK;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pfLock Points to the **FILELOCK** structure that specifies the starting position in the file and the number of bytes of the file to lock. This parameter is ignored if NULL is specified instead of a structure address.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION

Comments

The **DosFileLocks** function can both lock and unlock regions. The system unlocks any specified region before locking any other region. Locked regions can overlap, but if one region would entirely encompass another, the smaller region should be unlocked first. The **DosFileLocks** function can lock any part of a file. Attempting to lock bytes beyond the end of a file does not result in an error.

Example

This example opens the file *abc* and calls the **DosFileLocks** function to lock 100 bytes of the file, starting with byte number three. No other file may read or write to this range in the file until **DosFileLocks** is called to unlock the range or the file is closed. The same structure is used to lock the file and to unlock the file.

```
FILELOCK flock;
HFILE hf;
USHORT usAction;

/* open the file */

DosOpen("abc",                                  /* filename to open        */
    &hf,                                        /* address of file handle  */
    &usAction,                                  /* action taken            */
    100L,                                       /* size of new file        */
    FILE_NORMAL,                                /* file attribute          */
    FILE_OPEN,                                  /* open if file exists     */
    OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE    /* open mode            */
    0L);                                           /* reserved             */

flock.lOffset = 3L;                  /* offset to begin lock               */
flock.lRange = 100L;                 /* range to lock                      */
DosFileLocks(hf,                     /* handle of file to lock             */
    NULL,                            /* unlock range (NULL to disable)     */
    &flock);                         /* address of lock range              */
    .
    . /* other file processing occurs here */
    .
DosFileLocks(hf,                     /* handle of file to unlock           */
    &flock,                          /* address of unlock range            */
    NULL);                           /* lock range (NULL to disable)       */
```

See Also **DosDupHandle, DosExecPgm, DosOpen**

DosFindClose

USHORT DosFindClose(*hdir*)
HDIR *hdir*; /* handle of search directory */

The **DosFindClose** function closes the specified search-directory handle. The **DosFindFirst** and **DosFindNext** functions use the search-directory handle to locate files with names that match a given name.

The **DosFindClose** function is a family API function.

Parameters *hdir* Identifies the search directory. This handle must have been previously opened by using the **DosFindFirst** function.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_HANDLE

Example This example calls the **DosFindFirst** function to find all files that match "*.*". When **DosFindFirst** is finished, the handle is closed by calling the **DosFindClose** function.

```
HDIR hdir = 0xFFFF;
USHORT usSearchCount = 1;
FILEFINDBUF findbuf;
DosFindFirst("*.*", &hdir, FILE_NORMAL, &findbuf,
    sizeof(findbuf), &usSearchCount, 0L);
    .
    .
    .
DosFindClose(hdir);                  /* closes the search directory */
```

See Also **DosFindFirst, DosFindNext, DosSearchPath**

DosFindFirst

```
USHORT DosFindFirst(pszFileSpec, phdir, usAttribute, pfindbuf, usBufLen, pusSearchCount, ulReserved)
PSZ pszFileSpec;             /* pointer to string specifying pathname */
PHDIR phdir;                 /* pointer to variable for handle        */
USHORT usAttribute;          /* search attribute                      */
PFILEFINDBUF pfindbuf;       /* pointer to structure receiving result */
USHORT usBufLen;             /* length of result buffer               */
PUSHORT pusSearchCount;      /* pointer to variable for file count    */
ULONG ulReserved;            /* must be zero                          */
```

The **DosFindFirst** function searches a directory for the file or files whose filename and attributes match the specified filename and attributes. The function copies the name and directory information of the file to the **FILEFINDBUF** structure. The information returned is as accurate as the most recent call to the **DosClose** or **DosBufReset** function.

The **DosFindFirst** function is a family API function.

Parameters

pszFileSpec Points to a null-terminated string. This string must be a valid MS OS/2 pathname and may contain wildcard characters.

phdir Points to the variable that contains the handle of the directory to be searched.

If the *phdir* parameter is HDIR_SYSTEM, the system default search-directory handle is used. If it is HDIR_CREATE, the search directory that is used by the process is created, and the function copies the handle of this search directory to the variable pointed to by the *phDir* parameter. If the handle was created by a previous call to the **DosFindFirst** function, it can be used in subsequent calls to the **DosFindNext** function.

usAttribute Specifies the file attribute(s) of the file to be located. This parameter can be a combination of the following values:

Value	Meaning
FILE_NORMAL	Search for normal files.
FILE_READONLY	Search for read-only files.
FILE_HIDDEN	Search for hidden files.
FILE_SYSTEM	Search for system files.
FILE_DIRECTORY	Search for subdirectories.
FILE_ARCHIVED	Search for archived files.

pfindbuf Points to the **FILEFINDBUF** structure that receives the result of the search. The **FILEFINDBUF** structure has the following form:

```
typedef struct _FILEFINDBUF {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
    UCHAR  cchName;
    CHAR   achName[13];
} FILEFINDBUF;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

usBufLen Specifies the length (in bytes) of the structure pointed to by the *pfindbuf* parameter.

pusSearchCount Points to a variable that specifies the number of matching filenames to locate. The **DosFindFirst** function copies the number of filenames found to this parameter before returning.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BUFFER_OVERFLOW
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_NO_MORE_FILES
ERROR_NO_MORE_SEARCH_HANDLES
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND

Comments

The *pusSearchCount* parameter specifies the number of files to search for. The number of files whose information is copied is the number of files requested, the number of files whose information fits in the structure, or the number of files that exist, whichever is smallest. To receive information for more than one file, the *pfindbuf* parameter must point to a buffer that consists of consecutive **FILEFINDBUF** structures—for example, an array of structures. If the **DosFindFirst** function fails to find a match or cannot copy all of the information about the file to the structure, it returns an error.

The **DosFindFirst** function obtains a handle that can be used in subsequent calls to the **DosFindNext** function to specify the directory to search and the filename to search for. Each call to the **DosFindFirst** function automatically closes the handle of the search directory, if it has not been closed previously by using the **DosFindClose** function.

Currently, the maximum filename length is 13 bytes: up to 8 characters in the filename; 4 characters, including the period (.), in the filename extension; and the terminating null character. The maximum filename length will change in future versions of MS OS/2.

A search for read-only files, hidden files, system files, archived files, or subdirectories includes all normal files in addition to those matching the specified attribute.

Restrictions

In real mode, the following restriction applies to the **DosFindFirst** function:

- The *phdir* parameter must be set to HDIR_SYSTEM.

Example

This example uses the **DosFindFirst** function to find the file *abc.ext*. An error message is displayed if the file is not found.

```
HDIR hdir = HDIR_CREATE;
USHORT usSearchCount = 1;
FILEFINDBUF findbuf;
if (DosFindFirst("abc.ext",        /* filename to search for       */
        &hdir,                     /* address of directory handle */
        FILE_NORMAL,               /* type of files to search for */
        &findbuf,                  /* address of buffer           */
        sizeof(findbuf),           /* size of buffer              */
        &usSearchCount,            /* number of matching entries  */
        OL))                       /* reserved                    */
    VioWrtTTY("File not found\r\n", 16, 0);
else {
```

See Also

DosBufReset, DosClose, DosFindClose, DosFindNext, DosQFileMode, DosQFSInfo

DosFindNext

```
USHORT DosFindNext(hdir, pfindbuf, cbfindbuf, pusSearchCount)
HDIR hdir;                  /* handle of search directory                   */
PFILEFINDBUF pfindbuf;      /* pointer to structure receiving search result */
USHORT cbfindbuf;           /* length of result buffer                      */
PUSHORT pusSearchCount;     /* pointer to variable for file count           */
```

The **DosFindNext** function searches for the next file or group of files matching the specified filename and attributes. The function copies the name and directory information of the file to the **FILEFINDBUF** structure pointed to by the *pfindbuf* parameter. The information returned is as accurate as the most recent call to the **DosClose** or **DosBufReset** function.

The **DosFindNext** function is a family API function.

Parameters

hdir Identifies the search directory and the filename(s) to search for. This handle must have been created previously by using the **DosFindFirst** function.

pfindbuf Points to the **FILEFINDBUF** structure that receives the result of the search. The **FILEFINDBUF** structure has the following form:

```
typedef struct _FILEFINDBUF {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
    UCHAR  cchName;
    CHAR   achName[13];
} FILEFINDBUF;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

cbfindbuf Specifies the length (in bytes) of the structure pointed to by the *pfindbuf* parameter.

pusSearchCount Points to an unsigned variable that specifies the number of matching filenames to locate. The **DosFindNext** function copies the number of filenames found to the unsigned variable before returning.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BUFFER_OVERFLOW
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_NO_MORE_FILES
ERROR_NOT_DOS_DISK

Comments

The *pusSearchCount* parameter specifies the number of files to search for. The number of files whose information is copied is the number of files requested, the number of files whose information fits in the structure, or the number of files that exist, whichever is smallest. If you want to receive information for more than one file, the *pfindbuf* parameter must point to a buffer that consists of consecutive **FILEFINDBUF** structures—for example, an array of structures. If the **DosFindNext** function fails to find a match or cannot copy all of the information about the file to the structure, it returns an error.

Currently, the maximum filename length is 13 bytes: up to 8 characters in the filename; 4 characters, including the period (.), in the filename extension; and the terminating null character. The maximum filename length will change in future versions of MS OS/2.

Restrictions

In real mode, the following restriction applies to the **DosFindNext** function:

- The *hdir* parameter must be set to HDIR_SYSTEM.

Example

This example calls the **DosFindFirst** function to find all files matching "*.*", and then uses the **DosFindNext** function to display them one at a time:

```
HDIR hdir = 0xFFFF;
USHORT usSearchCount = 1;
FILEFINDBUF findbuf;
DosFindFirst("*.*", &hdir, 0x00, &findbuf, sizeof(findbuf),
    &usSearchCount, 0L);
do {
    VioWrtTTY(findbuf.achName, findbuf.cchName, 0);
    VioWrtTTY("\r\n", 2, 0);      /* cursor to next line     */
}
while (DosFindNext(hdir,          /* handle of directory     */
    &findbuf,                     /* address of buffer       */
    sizeof(findbuf),              /* length of buffer        */
    &usSearchCount)               /* number of files to find */
    == 0);                        /* while no error          */
```

See Also

DosBufReset, DosClose, DosFindClose, DosFindFirst, DosQFileMode, DosQFSInfo

DosFlagProcess

```
USHORT DosFlagProcess(pidProcess, fScope, usFlagNum, usFlagArg)
PID pidProcess;        /* identifier of process receiving flag */
USHORT fScope;         /* flag process or all processes        */
USHORT usFlagNum;      /* flag number                          */
USHORT usFlagArg;      /* flag argument                        */
```

The **DosFlagProcess** function generates a signal that is sent to the calling process. By default, the process ignores these signals, but it can respond to them by using the **DosSetSigHandler** function to define a signal handler. A process can also refuse event-flag signals, causing the **DosFlagProcess** function to return an error value.

Parameters

pidProcess Specifies the process identifier of the process that receives the flag.

fScope Specifies how many external event flags to set. If this parameter is FLGP_SUBTREE, the function sets the external event flags for the specified process and all of its child processes. If it is FLGP_PID, the function sets the event flag for only the specified process.

usFlagNum Specifies the number of the flag to set. This parameter can be one of the following values:

Value	Meaning
PFLG_A	Process flag A.
PFLG_B	Process flag B.
PFLG_C	Process flag C.

usFlagArg Specifies an argument to pass to the specified process.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_FLAG_NUMBER
ERROR_INVALID_FUNCTION
ERROR_INVALID_PROCID
ERROR_SIGNAL_REFUSED

Comments

The current signal cannot be accepted if a signal of the same type is already waiting to be processed.

Example

This example executes a process called *abc.exe*. It then calls the **DosFlagProcess** function to send the PFLG_A (process flag A) signal to that process.

```
CHAR achFailName[128];
RESULTCODES rescResults;
DosExecPgm(achFailName, sizeof(achFailName),
    EXEC_ASYNCH, "abc ", 0, &rescResults, "abc.exe");
    .
    .
    .
DosFlagProcess(rescResults.codeTerminate, /* process identifier */
    FLGP_SUBTREE,                 /* notifies the entire subtree */
    PFLG_A,                       /* sends process flag A        */
    1);                           /* value to send process       */
```

See Also

DosExecPgm, DosSetSigHandler

DosFreeModule

```
USHORT DosFreeModule(hmod)
HMODULE hmod;    /* module handle */
```

The **DosFreeModule** function frees the specified dynamic-link module. After a process has freed a module, any function addresses the process may have retrieved from the module are no longer valid; a protection fault occurs if these functions are called.

Parameters

hmod Identifies the dynamic-link module to free. This handle must have been created previously by using the **DosLoadModule** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INTERRUPT
ERROR_INVALID_HANDLE

Comments

If other processes have loaded the module and not yet freed it, the module remains in system memory for those processes. The system does not remove a module from memory until it is no longer used by any process.

See Also

DosLoadModule

DosFreeSeg

```
USHORT DosFreeSeg(sel)
SEL sel;    /* segment selector */
```

The **DosFreeSeg** function frees the specified memory segment. The function accepts selectors for memory segments, shared-memory segments, huge-memory segments, and aliased code segments. **DosFreeSeg** frees a shared-memory segment after the segment is freed by the last process accessing it. **DosFreeSeg** frees the code-segment selector for aliased code segments, but the corresponding data-segment selector remains valid until it is freed.

The **DosFreeSeg** function is a family API function.

Parameters

sel Specifies the selector of the segment to free.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_ACCESS_DENIED

Restrictions

In real mode, the following restriction applies to the **DosFreeSeg** function:

- A code-segment selector (created by using the **DosCreateCSAlias** function) and the corresponding data-segment selector are the same. Freeing one frees both.

Example

This example allocates three segments of memory, then calls the **DosFreeSeg** function to free the memory:

```
SEL sel;
DosAllocHuge(3, 200, &sel, 5, SEG_NONSHARED);
    .
    .
    .
DosFreeSeg(sel);
```

See Also

DosAllocHuge, DosAllocSeg, DosAllocShrSeg, DosCreateCSAlias

DosFSRamSemClear

USHORT DosFSRamSemClear(*pdosfsrs***)**
PDOSFSRSEM *pdosfsrs*; /* pointer to structure for semaphore */

The **DosFSRamSemClear** function releases ownership of a fast-safe RAM semaphore.

Parameters

pdosfsrs Points to the **DOSFSRSEM** structure containing the information about a fast-safe RAM semaphore. The **DOSFSRSEM** structure has the following form:

```
typedef struct _DOSFSRSEM {
    USHORT cb;
    PID    pid;
    TID    tid;
    USHORT cUsage;
    USHORT client;
    ULONG  sem;
} DOSFSRSEM;
```

For more information, see Chapter 4, "Types, Macros, Structures."

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **DosFSRamSemClear** function is used to release a semaphore obtained by using the **DosFSRamSemRequest** function. If the semaphore-use count for the current thread is zero, the semaphore is cleared and any threads that are blocked on the semaphore are restarted.

DosFSRamSemClear cannot be issued against a fast-safe RAM semaphore that is owned by another thread.

See Also

DosFSRamSemRequest

DosFSRamSemRequest

USHORT DosFSRamSemRequest(*pdosfsrs***,** *lTimeout***)**
PDOSFSRSEM *pdosfsrs*; /* pointer to structure for semaphore */
LONG *lTimeout*; /* time to wait for semaphore */

The **DosFSRamSemRequest** function obtains a fast-safe RAM semaphore and records the current owner for potential cleanup by a **DosExitList** function.

Parameters

pdosfsrs Points to the **DOSFSRSEM** structure containing information about a fast-safe RAM semaphore. The **DOSFSRSEM** structure has the following form:

```
typedef struct _DOSFSRSEM {
    USHORT cb;
    PID    pid;
    TID    tid;
    USHORT cUsage;
    USHORT client;
    ULONG  sem;
} DOSFSRSEM;
```

For more information, see Chapter 4, "Types, Macros, Structures."

lTimeout Specifies how long to wait for the semaphore to become available. If the value is greater then zero, this parameter specifies the number of milliseconds to wait before returning. If the value is SEM_IMMEDIATE_RETURN, the function returns immediately. If the value is SEM_INDEFINITE_WAIT, the function waits indefinitely.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

When the **DosFSRamSemRequest** function is called, it checks the status of the semaphore. If the semaphore is not owned, **DosFSRamSemRequest** sets it to owned, increases the use count, and returns immediately to the calling function. If the semaphore is owned, **DosFSRamSemRequest** may block the thread until the semaphore is not owned, then try again. The *lTimeout* parameter is used to place an upper limit on the amount of time to block before returning.

When the thread is finished with the protected resource, it calls the **DosFSRamSemClear** function. **DosFSRamSemClear** decreases the use count and, if the count is zero, sets the semaphore to unowned and starts any threads that were blocked while waiting for the semaphore.

Recursive requests for fast-safe RAM semaphores are supported by a use count of the number of times the owning process has issued a **DosFSRamSemRequest** function without issuing a corresponding **DosFSRamSemClear** function.

The **DosFSRamSemRequest** function does not return unless the specified semaphore remains clear long enough for the calling thread to obtain it.

Fast-safe RAM semaphores operate by using the **DOSFSRSEM** structure. Before the initial call to the **DosFSRamSemRequest** function, this structure must be initialized to zero and the **cb** field must be set to 14. The **client** field is provided to allow the calling process a means of identifying which resource is currently owned by the owner of the semaphore. This field is initialized to zero when a fast-safe RAM semaphore is first acquired. The owning process may use this field to describe the resource currently being accessed. The values in the **client** field may be useful to an **DosExitList** function handler in determining the appropriate cleanup action.

When a process terminates that owns a fast-safe RAM semaphore, the **DosExitList** functions of that process (if any) will be given control. If important resources are protected by fast-safe RAM semaphores, the **DosExitList** function should call the **DosFSRamSemRequest** function to gain ownership of these semaphores. When called during the processing of **DosExitList** termination functions, the **DosFSRamSemRequest** function will examine the indicated fast-safe RAM semaphore and, if it is owned by the active process, force the identifier of the owning thread to be equal to the identifier of the current thread and set the use count to one. This allows the **DosExitList** function to be used without requiring any handling instructions for fast-safe RAM semaphores. When the execution of the **DosExitList** function is finished, it should call the **DosFSRamSemClear** function.

Except for the **client** field, the calling process should not modify any fields in the **DOSFSRSEM** structure after the **DosFSRamSemRequest** function returns.

See Also **DosExitList, DosFSRamSemClear**

DosGetCollate

```
USHORT DosGetCollate(cbBuf, pctryc, pchBuf, pcbTable)
USHORT cbBuf;               /* size of buffer                                 */
PCOUNTRYCODE pctryc;        /* pointer to structure containing country code  */
PCHAR pchBuf;               /* pointer to buffer for table                    */
PUSHORT pcbTable;           /* pointer to variable receiving table length     */
```

The **DosGetCollate** function retrieves the collating-sequence table for the given country code and code-page identifier. The collating-sequence table is a character array with 256 elements in which each element specifies the sorting weight of the corresponding character. (The sorting weight is the value used to determine if a character appears before or after another character in a sorted list.) Sorting weights and character values are not necessarily the same—for example, in a given character set, the sorting weights for the letters A and B might be 1 and 2, even though their character values are 65 and 66.

The **DosGetCollate** function copies the collating-sequence table from the *country.sys* file to a buffer. If the buffer is too small to hold all the information, **DosGetCollate** truncates the information. If the buffer is larger than the information, **DosGetCollate** fills any remaining bytes with zeros.

The **DosGetCollate** function is a family API function.

Parameters

cbBuf Specifies the size (in bytes) of the buffer that receives the collating-sequence table.

pctryc Points to the **COUNTRYCODE** structure that contains the country code and the code-page identifier used to retrieve the collating-sequence table. The **COUNTRYCODE** structure has the following form:

```
typedef struct _COUNTRYCODE {
    USHORT country;
    USHORT codepage;
} COUNTRYCODE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pchBuf Points to the buffer that receives the collating-sequence table.

pcbTable Points to the variable that receives the number of bytes copied to the buffer.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The MS OS/2 **sort** command uses the **DosGetCollate** function to sort text according to the collating-sequence table.

Restrictions

In real mode, the following restriction applies to the **DosGetCollate** function:

- There is no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

See Also

DosCaseMap, **DosGetCtryInfo**

DosGetCp

```
USHORT DosGetCp(cbBuf, pusBuf, pcbCodePgLst)
USHORT cbBuf;              /* number of bytes in buffer for list          */
PUSHORT pusBuf;            /* pointer to buffer receiving list            */
PUSHORT pcbCodePgLst;      /* pointer to variable receiving list length  */
```

The **DosGetCp** function retrieves a list that contains the current code page for the process and all prepared system code pages. The code-page list consists of one or more 16-bit values, each value representing a code-page identifier. The first value in the list is the identifier for the process's current code page. A process can set its current code page by using the **DosSetCp** function. Otherwise, the process inherits its current code page from its parent process.

The **DosGetCp** function copies the code-page list to a buffer. If the buffer is too small to hold all the information, **DosGetCp** truncates the information. If the buffer is larger than the information, **DosGetCp** fills any remaining bytes with zeros.

Parameters

cbBuf Specifies the length (in bytes) of the buffer for the code-page list.

pusBuf Points to the buffer that receives the code-page list.

pcbCodePgLst Points to the variable that receives the number of bytes copied to the code-page list.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The code-page identifier can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

See Also

DosSetCp

DosGetCtryInfo

```
USHORT DosGetCtryInfo(cbBuf, pctryc, pctryi, pcbCountryInfo)
USHORT cbBuf;                /* length of data area                          */
PCOUNTRYCODE pctryc;         /* pointer to structure containing country info. */
PCOUNTRYINFO pctryi;         /* pointer to structure receiving country info.  */
PUSHORT pcbCountryInfo;      /* pointer to variable for number of bytes      */
```

The **DosGetCtryInfo** function retrieves a copy of the country-dependent formatting information for the specified country code and code-page identifier. Country-dependent formatting information defines the symbols and formats used to express currency values, dates, times, and numbers in a given country.

The **DosGetCtryInfo** function copies the information from the *country.sys* file to the **COUNTRYINFO** structure. If this structure is too small to hold all the information, **DosGetCtryInfo** truncates the information. If the structure is larger than the information, the function fills any remaining bytes with zeros.

The **DosGetCtryInfo** function is a family API function.

Parameters

cbBuf Specifies the size (in bytes) of the **COUNTRYINFO** structure.

pctryc Points to the **COUNTRYCODE** structure that contains the country code and the code-page identifier used to retrieve country-dependent information. The **COUNTRYCODE** structure has the following form:

```
typedef struct _COUNTRYCODE {
    USHORT country;
    USHORT codepage;
} COUNTRYCODE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pctryi Points to the **COUNTRYINFO** structure that receives the country-dependent formatting information. The **COUNTRYINFO** structure has the following form:

```
typedef struct _COUNTRYINFO {
    USHORT country;
    USHORT codepage;
    USHORT fsDateFmt;
    CHAR   szCurrency[5];
    CHAR   szThousandsSeparator[2];
    CHAR   szDecimal[2];
    CHAR   szDateSeparator[2];
    CHAR   szTimeSeparator[2];
    UCHAR  fsCurrencyFmt;
    UCHAR  cDecimalPlace;
    UCHAR  fsTimeFmt;
    USHORT abReserved1[2];
    CHAR   szDataSeparator[2];
    USHORT abReserved2[5];
} COUNTRYINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pcbCountryInfo Points to the variable that receives the number of bytes of information copied to the **COUNTRYINFO** structure.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_NLS_BAD_TYPE
ERROR_NLS_NO_COUNTRY_FILE
ERROR_NLS_NO_CTRY_CODE
ERROR_NLS_OPEN_FAILED
ERROR_NLS_TABLE_TRUNCATED
ERROR_NLS_TYPE_NOT_FOUND

Restrictions

In real mode, the following restriction applies to the **DosGetCtryInfo** function:

- There is no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

DosGetDateTime

USHORT DosGetDateTime(*pdateTime*)
PDATETIME *pdateTime*; /* pointer to structure for date and time */

The **DosGetDateTime** function retrieves the current date and time. Although MS OS/2 maintains the current date and time, any process can change the date and time by using the **DosSetDateTime** function; as a result, the current date and time are as accurate as the most recent call to the **DosSetDateTime** function.

The **DosGetDateTime** function is a family API function.

Parameters

pdateTime Points to the **DATETIME** structure that receives the date and time information. The **DATETIME** structure has the following form:

```
typedef struct _DATETIME {
    UCHAR   hours;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    SHORT   timezone;
    UCHAR   weekday;
} DATETIME;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

A process can also retrieve the current date and time by using the **DosGetInfoSeg** function. However, **DosGetInfoSeg** is available only to programs that run with MS OS/2.

Example

This example calls the **DosGetDateTime** function repeatedly until the time is 9:30:

```
DATETIME date;
do                                   /* do until 9:30 */
    DosGetDateTime(&date);
while (!(date.hours == 9 && date.minutes == 30))
```

See Also

DosGetInfoSeg, DosSetDateTime

DosGetDBCSEv

```
USHORT DosGetDBCSEv(cbBuf, pctryc, pchBuf)
USHORT cbBuf;                /* length of buffer                    */
PCOUNTRYCODE pctryc;         /* pointer to structure for country code */
PCHAR pchBuf;                /* pointer to buffer for DBCS information */
```

The **DosGetDBCSEv** function retrieves the double-byte character set (DBCS) environment vector for the given country code and code-page identifier.

The **DosGetDBCSEv** function is a family API function.

Parameters

cbBuf Specifies the size (in bytes) of the buffer that receives the DBCS environment vector.

pctryc Points to the **COUNTRYCODE** structure that contains the country code and code-page identifier used to retrieve the DBCS environment vector. The **COUNTRYCODE** structure has the following form:

```
typedef struct _COUNTRYCODE {
    USHORT country;
    USHORT codepage;
} COUNTRYCODE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pchBuf Points to the buffer that receives the country-dependent DBCS environment vector.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_NLS_BAD_TYPE
ERROR_NLS_NO_COUNTRY_FILE
ERROR_NLS_NO_CTRY_CODE
ERROR_NLS_OPEN_FAILED
ERROR_NLS_TABLE_TRUNCATED
ERROR_NLS_TYPE_NOT_FOUND

Comments

The DBCS environment vector defines the first and last values in the ranges for the DBCS lead-byte and second-byte values.

The **DosGetDBCSEv** function copies the information from the *country.sys* file to a buffer. The first two bytes in the environment vector specify the first and last values in the range for the DBCS lead-byte values. All subsequent pairs of bytes (except for the last two bytes) specify the first and last values in the ranges for DBCS second-byte values. The last two bytes are both set to zero. The form of the information is similar to the following:

```
CHAR first1, last1;
CHAR first2, last2;
     .
     .
     .
CHAR firstn, lastn;
CHAR firstend=0, lastend=0;
```

If the buffer is too small to hold all of the information, the **DosGetDBCSEv** function truncates the information. To avoid this, make sure the buffer is at least ten bytes long. You can verify that all information has been copied by checking the last two bytes to make sure they are zeros. If the structure is larger than the information, the function fills any remaining bytes with zeros.

Restrictions

In real mode, the following restriction applies to the **DosGetDBCSEv** function:

- There is no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

See Also

DosCaseMap, DosGetCollate, DosGetCp, DosGetCtryInfo, DosSetCp, VioGetCp, VioSetCp

■ DosGetEnv

```
USHORT DosGetEnv(pselEnviron, pusOffsetCmd)
PUSHORT pselEnviron;      /* pointer to variable for selector */
PUSHORT pusOffsetCmd;     /* pointer to variable for offset   */
```

The **DosGetEnv** function retrieves the address of the process's environment and an offset into the environment where the command line is stored that was used to start the process. This offset can be used to retrieve command-line arguments.

The environment is one or more null-terminated strings that name and define the environment variables available to the current process. The command-line string is a single null-terminated string that is a copy of the command line that was used to run the process.

The **DosGetEnv** function is a family API function.

Parameters

pselEnviron Points to the variable that receives the environment's segment selector. The environment begins in the first byte of the segment identified by this parameter.

pusOffsetCmd Points to the variable that receives the offset from the beginning of the specified segment to the beginning of the command line.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_ACCESS

Comments

Each string in the environment has the following form:

stringname=value

Each environment string ends with a null character. The last string is followed by an additional null character to indicate the end of the environment. The two null characters are followed by the command-line string.

The **DosGetEnv** function is typically used to retrieve the command-line arguments. Although **DosGetEnv** can be used to retrieve a single environment variable, an easier way to do this is to call the **DosScanEnv** function.

Example

This example calls the **DosGetEnv** function to retrieve the selector to the environment and the offset to the argument table within the environment. The *pszEnviron* parameter points to the beginning of the environment, and the *pszArgument* parameter points to the beginning of the argument table.

```
PSZ pszEnviron, pszArgument;
SEL selEnviron;
USHORT usOffsetCmd;
DosGetEnv(&selEnviron, &usOffsetCmd);
pszEnviron = MAKEP(selEnviron, 0);
pszArgument = MAKEP(selEnviron, usOffsetCmd);
```

See Also

DosExecPgm, DosScanEnv

■ DosGetHugeShift

USHORT DosGetHugeShift(*pusShiftCount***)**
PUSHORT *pusShiftCount*; /* pointer to variable receiving shift count */

The **DosGetHugeShift** function retrieves the shift count used to compute the segment-selector offset for huge memory segments. (Huge memory segments are allocated by using the **DosAllocHuge** function.) The shift count represents a multiple of two, so the segment-selector offset is equal to the value 1 shifted left by the shift count. For example, the segment-selector offset is eight if the shift count is three.

The **DosGetHugeShift** function is a family API function.

Parameters

pusShiftCount Points to the variable that receives the shift count.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

DosAllocHuge

■ DosGetInfoSeg

USHORT DosGetInfoSeg(*pselGlobalSeg*, *pselLocalSeg***)**
PSEL *pselGlobalSeg*; /* pointer to variable for global selector */
PSEL *pselLocalSeg*; /* pointer to variable for local selector */

The **DosGetInfoSeg** function retrieves segment selectors for the global and local information segments. These read-only information segments contain general information about the system and the process. The global information segment is accessible only to all processes. The local information segment is accessible only to the current process.

Parameters

pselGlobalSeg Points to the **GINFOSEG** structure that contains global information. The **GINFOSEG** structure has the following form:

```
typedef struct _GINFOSEG {
    ULONG   time;
    ULONG   msecs;
    UCHAR   hour;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    USHORT  timezone;
    USHORT  cusecTimerInterval;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    UCHAR   weekday;
    UCHAR   uchMajorVersion;
    UCHAR   uchMinorVersion;
    UCHAR   chRevisionLetter;
    UCHAR   sgCurrent;
    UCHAR   sgMax;
    UCHAR   cHugeShift;
    UCHAR   fProtectModeOnly;
    USHORT  pidForeground;
    UCHAR   fDynamicSched;
    UCHAR   csecMaxWait;
    USHORT  cmsecMinSlice;
    USHORT  cmsecMaxSlice;
    USHORT  bootdrive;
    UCHAR   amecRAS[32];
    UCHAR   csgWindowableVioMax;
    UCHAR   csgPMMax;
} GINFOSEG;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pselLocalSeg Points to the **LINFOSEG** structure that contains local information. The **LINFOSEG** structure has the following form:

```
typedef struct _LINFOSEG {
    PID     pidCurrent;
    PID     pidParent;
    USHORT  prtyCurrent;
    TID     tidCurrent;
    USHORT  sgCurrent;
    UCHAR   rfProcStatus;
    UCHAR   dummy1;
    BOOL    fForeground;
    UCHAR   typeProcess;
    UCHAR   dummy2;
    SEL     selEnvironment;
    USHORT  offCmdLine;
    USHORT  cbDataSegment;
    USHORT  cbStack;
    USHORT  cbHeap;
    HMODULE hmod;
    SEL     selDS;
} LINFOSEG;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Example

This example calls the **DosGetInfoSeg** function to retrieve the selector of a system global segment, converts the segment selector into a pointer to a structure, and checks to determine if the current day of the week is Monday:

```
SEL selGlobalSeg, selLocalSeg;
GINFOSEG FAR *pgis;
DosGetInfoSeg(&selGlobalSeg, &selLocalSeg);
pgis = MAKEPGINFOSEG(selGlobalSeg);
if (pgis->weekday == 1) {
        .
        . /* this code is executed only on a Monday */
        .
```

See Also

DosGetDateTime

■ DosGetMachineMode

USHORT DosGetMachineMode(*pbMachineMode***)**
PBYTE *pbMachineMode*; /* pointer to variable for machine mode */

The **DosGetMachineMode** function retrieves the current machine mode—that is, whether the current mode is real or protected.

The **DosGetMachineMode** function is a family API function.

Parameters

pbMachineMode Points to the variable that receives the machine mode. If this parameter is MODE_REAL, the current mode is real mode, 808*x* or 80*x*86. If this parameter is MODE_PROTECTED, the current mode is protected mode, 80*x*86.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

The **DosGetMachineMode** function allows a program that is running in real mode to avoid calling functions that are not available when it is in real mode. The MS OS/2 functions that are available in both real and protected modes are listed in the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

Example

This example calls the **DosGetMachineMode** function and displays the machine mode under which the current process is running:

```
BYTE bMode;
DosGetMachineMode(&bMode);
if (bMode == MODE_PROTECTED)
    VioWrtTTY("Protected mode\r\n", 16, 0);
else
    VioWrtTTY("Real mode\r\n", 11, 0);
```

DosGetMessage

```
USHORT DosGetMessage(ppchVTable, usVCount, pchBuf, cbBuf, usMsgNo, pszFileName, pcbMsg)
PCHAR FAR * ppchVTable;   /* pointer to table of pointers to strings */
USHORT usVCount;          /* number of pointers in table             */
PCHAR pchBuf;             /* pointer to buffer receiving message     */
USHORT cbBuf;             /* number of bytes in buffer               */
USHORT usMsgNo;           /* message number to retrieve              */
PSZ pszFileName;          /* name of file containing message         */
PUSHORT pcbMsg;           /* number of bytes in returned message     */
```

The **DosGetMessage** function retrieves a message from the specified system-message file. **DosGetMessage** may insert one or more strings into the body of the message as it retrieves the message.

The **DosGetMessage** function is a family API function.

Parameters

ppchVTable Points to a table of pointers to substitution strings. Each entry in the table points to a null-terminated string to be inserted into the message. Up to nine pointers can be given.

usVCount Specifies the number of pointers in the table. This parameter can be any value from 0 through 9. If this parameter is zero, the *ppchVTable* parameter is ignored. If it is greater than 9, the **DosGetMessage** function returns an error indicating that the *usVCount* parameter is out of range.

pchBuf Points to the buffer that receives the requested message.

cbBuf Specifies the length (in bytes) of the buffer.

usMsgNo Specifies the message number for the requested message.

pszFileName Points to a null-terminated string that specifies the MS OS/2 path and filename of the message file that contains the message.

pcbMsg Points to the variable that receives the number of bytes copied to the buffer.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_FILE_NOT_FOUND
ERROR_MR_INV_IVCOUNT
ERROR_MR_INV_MSGF_FORMAT
ERROR_MR_MID_NOT_FOUND
ERROR_MR_MSG_TOO_LONG
ERROR_MR_UN_ACC_MSGF
ERROR_MR_UN_PERFORM

Comments

To retrieve the requested message, the **DosGetMessage** function first searches the process's message segment, if there is one. If it cannot find the specified message, the function then searches the specified message file. If no drive or path is specified in the filename, **DosGetMessage** searches the system root directory for the message file, then searches the current directory on the current drive. The **DosGetMessage** function may also search the directories specified by the commands **append** (in real mode) and **dpath** (in protected mode) for the given message file.

When the **DosGetMessage** function finds a message, it copies the message to the buffer pointed to by the *pchBuf* parameter. As it copies the message, **DosGetMessage** replaces any symbol in the form %*x* (where *x* is a digit from 1 through 9) with one of the strings pointed to in the table pointed to by the *ppchVTable* parameter. For example, **DosGetMessage** replaces all symbols in the form %1 with the string pointed to by the first pointer in the table. If there is no corresponding string in the table, **DosGetMessage** copies the %*x* symbol, unchanged, to the buffer.

The %*x* symbols used in a message are not necessarily enclosed in spaces. If you want spaces, you may need to supply them as part of your substitution strings.

If the message is too long to fit in the buffer, the **DosGetMessage** function truncates the message and returns an error code.

If the **DosGetMessage** function cannot retrieve a message because of a direct-access-storage-device (DASD) hard error or because it cannot find the message file, it places a default message in the buffer. This can occur when an invalid parameter is specified—for example, an invalid *usMsgNo* parameter or an invalid *usVCount* parameter; when the **DosGetMessage** function cannot read the system-message file—for example, when a DASD error occurs or when format of the message file is invalid; or when the **DosGetMessage** function cannot find the system-message file. The **DosGetMessage** function retrieves messages that have been prepared previously by using the **mkmsgf** utility to create a message file. **DosGetMessage** also retrieves messages that have been added to the message segment of the program's executable file by using the **msgbind** utility. It is irrelevant to the process that calls the **DosGetMessage** function whether **DosGetMessage** retrieves messages from the message segment or from the message file. In either case, the function uses the *usMsgNo* and *pszFileName* parameters to locate the message. For more information on the **mkmsgf** and **msgbind** utilities, see *Microsoft Operating System/2 Programming Tools*.

Restrictions

In real mode, the following restriction applies to the **DosGetMessage** function:

- There is no method of identifying the boot drive.

See Also

DosInsMessage, DosPutMessage

DosGetModHandle

```
USHORT DosGetModHandle(pszModName, phMod)
PSZ pszModName;         /* module name                          */
PHMODULE phMod;         /* pointer to variable receiving module handle */
```

The **DosGetModHandle** function retrieves the handle of a dynamic-link module. The **DosGetModHandle** function is typically used to make sure that a module has been loaded into memory. If the module has not been loaded, the function returns an error value.

Parameters

pszModName Points to a null-terminated string that specifies the MS OS/2 filename of the module. The *.dll* filename extension is used for dynamic-link libraries.

phMod Points to the variable that receives the module handle.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INTERRUPT
ERROR_MOD_NOT_FOUND

Comments

The module name specified by the *pszModName* parameter must match the name of the module that is already loaded. Otherwise, an error value is returned.

See Also

DosFreeModule, DosGetModName, DosLoadModule

DosGetModName

```
USHORT DosGetModName(hmod, cbBuf, pchBuf)
HMODULE hmod;       /* module handle                           */
USHORT cbBuf;       /* number of bytes in buffer               */
PCHAR pchBuf;       /* pointer to buffer receiving module name */
```

The **DosGetModName** function retrieves the drive, path, and filename of the specified module.

Parameters

hmod Identifies the dynamic-link module. This handle must have been created previously by using the **DosLoadModule** function.

cbBuf Specifies the maximum length (in bytes) of the buffer that receives the the information about the module.

pchBuf Points to the buffer that receives the module's drive, path, and filename.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_LENGTH
ERROR_INTERRUPT
ERROR_INVALID_HANDLE

Comments

The **DosGetModName** function returns an error if there is not enough room in the buffer for the drive, path, and filename.

When a function within a dynamic-link library is called, or when the dynamic-link library initializes itself, the **di** register contains the module handle for the current process.

See Also

DosFreeModule, DosGetModHandle, DosLoadModule, DosMonOpen

DosGetPID

USHORT DosGetPID(*ppidi***)**
PPIDINFO *ppidi*; /* pointer to structure receiving identifiers */

The **DosGetPID** function retrieves the process, thread, and parent-process identifiers for the current process.

Parameters

ppidi Points to the **PIDINFO** structure that receives the process identifiers. The **PIDINFO** structure has the following form:

```
typedef struct _PIDINFO {
    PID pid;
    TID tid;
    PID pidParent;
} PIDINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

DosExecPgm, DosGetPPID

DosGetPPID

USHORT DosGetPPID(*pidChild, ppidParent***)**
USHORT *pidChild*; /* process identifier of child process */
PUSHORT *ppidParent*; /* point to variable for parent-process identifier */

The **DosGetPPID** function retrieves the process identifier of a parent process.

Parameters

pidChild Specifies the process identifier of the child process.

ppidParent Points to the variable that receives the process identifier of the parent process.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_PROCID

See Also

DosGetPID

DosGetProcAddr

```
USHORT DosGetProcAddr(hmod, pszProcName, ppfnProcAddress)
HMODULE hmod;              /* handle of module                        */
PSZ pszProcName;           /* pointer to module-name string           */
PPFN ppfnProcAddress;      /* pointer to variable for procedure address */
```

The **DosGetProcAddr** function retrieves the address of a procedure in a specified dynamic-link module. This address can then be used to call the procedure.

Parameters

hmod Identifies the dynamic-link module. This handle must have been created previously by using the **DosLoadModule** function.

pszProcName Points to a null-terminated string that specifies the procedure name to retrieve. If this string starts with a number sign (#), the remaining part of the string is treated as an ASCII ordinal. Alternately, if the selector portion of the pointer is zero, the offset portion of the pointer is an explicit entry number (an ordinal) within the dynamic-link module.

ppfnProcAddress Points to the variable that receives the procedure address.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INTERRUPT
ERROR_INVALID_HANDLE
ERROR_PROC_NOT_FOUND

Comments

Although the **DosGetProcAddr** function can be used to retrieve procedure addresses from the DOSCALLS dynamic-link module, these procedures are available through ordinal values only. If you attempt to retrieve a procedure address from the DOSCALLS module by using a procedure name, **DosGetProcAddr** returns an error.

Example

This example calls the **DosLoadModule** function to load the dynamic-link module *qhdll.dll*. It then calls the **DosGetProcAddr** function to retrieve the address of the BOXMESSAGE function that is defined in the module and calls the **DosFreeModule** function to free the dynamic-link module. (This example is accurate if *qhdll.dll* exists in a directory defined by the **libpath** parameter of the *config.sys* file, and if *qhdll.dll* contains the BOXMESSAGE function that uses the Pascal calling convention.)

```
CHAR achFailName[128];
HMODULE hmod;
VOID (PASCAL FAR *pfnBoxMsg) (PSZ, BYTE, BYTE, SHANDLE, SHANDLE, BOOL);

DosLoadModule(achFailName, sizeof(achFailName), "qhdll", &hmod);
DosGetProcAddr(hmod,          /* module handle                */
    "BOXMESSAGE",             /* name of function             */
    &pfnBoxMsg);              /* variable for function address */
pfnBoxMsg("Hello World", 0x30, 1, 0, 0);
DosFreeModule(hmod);
```

See Also

DosFreeModule, DosGetModName, DosLoadModule

DosGetPrty

```
USHORT DosGetPrty(usScope, pusPriority, pid)
USHORT usScope;        /* thread priority in current process/another process */
PUSHORT pusPriority;   /* pointer to variable for priority                   */
USHORT pid;            /* process or thread identifier                       */
```

The **DosGetPrty** function retrieves the scheduling priority of a specified thread in the current process or the priority of thread 1 in a specified process.

Parameters

usScope Specifies whether to retrieve the priority for a thread in the current process or the priority of thread 1 in some other process.

If the *usScope* parameter is PRTYS_PROCESS, the **DosGetPrty** function retrieves the priority of thread 1 for the process specified by the *pid* parameter. If thread 1 for that process has terminated, the **DosGetPrty** function returns an error value.

If the *usScope* parameter is PRTYS_THREAD, the function retrieves the priority of the thread specified by the *pid* parameter.

pusPriority Points to the variable that receives the scheduling priority of the specified thread. The high-order byte is set to the priority class; the low-order byte is set to the priority level.

pid Specifies a process or thread identifier, depending on the value of the *usScope* parameter. If the *pid* parameter is 0x0000, the **DosGetPrty** function retrieves the priority for the current process or thread.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_PROCID
ERROR_INVALID_SCOPE
ERROR_INVALID_THREADID

See Also

DosSetPrty

DosGetResource

```
USHORT DosGetResource(hmod, idType, idName, psel)
HMODULE hmod;     /* module handle                             */
USHORT idType;    /* resource-type identifier                  */
USHORT idName;    /* resource-name identifier                  */
PSEL psel;        /* pointer to variable for resource selector */
```

The **DosGetResource** function retrieves the specified resource from a specified executable file. The function allocates a segment, copies the resource into the segment, and returns the segment selector. A process can use this segment selector to access the resource directly.

Parameters

hmod Identifies the module that contains the resource. This parameter can be either the module handle returned by the **DosLoadModule** function or NULL for the application's module.

idType Specifies the type of resource to retrieve.

idName Specifies the name of the resource to retrieve.

psel Points to the variable that receives the selector of the segment containing the resource.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_CANT_FIND_RESOURCE
ERROR_INVALID_MODULE
ERROR_INVALID_SELECTOR

Comments

The following list describes the predefined types that can be used for the *idType* parameter:

Type	Meaning
RT_ACCELTABLE	Accelerator tables
RT_BITMAP	Bitmap
RT_CHARTBL	Glyph to character tables
RT_DIALOG	Dialog template
RT_DISPLAYINFO	Screen-display information
RT_FONT	Font
RT_FONTDIR	Font directory
RT_KEYTBL	Key to UGL tables
RT_MENU	Menu template
RT_MESSAGE	Error-message tables
RT_POINTER	Mouse-pointer shape
RT_RCDATA	Binary data
RT_STRING	String tables
RT_VKEYTBL	Key to virtual-key tables

See Also

DosLoadModule

■ DosGetSeg

```
USHORT DosGetSeg(sel)
SEL sel;    /* selector of shared memory segment */
```

The **DosGetSeg** function obtains access to the shared memory segment identified by a specified segment selector. Although a process can receive the selector for a shared memory segment from another process, it cannot use the selector to access the segment until it uses the **DosGetSeg** function.

Parameters

sel Specifies the selector for the shared memory segment.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments **DosGetSeg** obtains access only to shared memory segments created by using the **DosAllocSeg** function with the *fAlloc* parameter set to SEG_GETTABLE.

See Also **DosAllocSeg, DosGetShrSeg, DosGiveSeg**

DosGetShrSeg

```
USHORT DosGetShrSeg(pszName, psel)
PSZ pszName;    /* pointer to memory-segment name */
PSEL psel;      /* pointer to variable for selector */
```

The **DosGetShrSeg** function retrieves a selector to a shared memory segment. The shared segment must have been allocated previously by another process. The function increases the segment's reference count by one to indicate that the segment is in use. The process receiving the new selector may use it to obtain access to the shared memory segment.

Parameters

pszName Points to a null-terminated string that identifies the shared memory segment. This string must have the following form:

\sharemem*pszName*

The string name, *pszName*, must have the same format as an MS OS/2 filename and must be unique.

psel Points to the variable that receives the new selector for the shared memory segment.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_FILE_NOT_FOUND
ERROR_INVALID_HANDLE
ERROR_TOO_MANY_OPEN_FILES

See Also **DosAllocShrSeg, DosFreeSeg, DosGetSeg**

DosGetVersion

```
USHORT DosGetVersion(pusVersion)
PUSHORT pusVersion;    /* pointer to variable receiving version number */
```

The **DosGetVersion** function retrieves the operating system's version number. For MS OS/2, version 1.1, both the major and minor version numbers are 10.

The **DosGetVersion** function is a family API function.

Parameters

pusVersion Points to the variable that receives the version number. The high-order byte is set to the major version number; the low-order byte is set to the minor version number.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Example

This example retrieves the version number and displays the major version number:

```
USHORT usVersion;
CHAR ch;

DosGetVersion(&usVersion);
ch = (LOBYTE(usVersion) / 10) + '0'; /* gets major version number */
VioWrtTTY("You are using MS OS/2 version ", 30, 0);
VioWrtTTY(&ch, 1, 0);
VioWrtTTY("\r\n", 2, 0);
```

See Also

DosQSysInfo

DosGiveSeg

```
USHORT DosGiveSeg(sel, pidProcess, pselRecipient)
SEL sel;                /* selector of shared memory segment      */
PID pidProcess;         /* process identifier of recipient        */
PSEL pselRecipient;     /* pointer to variable for selector of recipient */
```

The **DosGiveSeg** function creates a new segment selector for a shared memory segment. The new selector can then be used by another process to access the shared memory segment.

The process that creates the new segment selector is responsible for passing the selector to any process that uses the segment.

Parameters

sel Specifies the segment selector of the shared memory segment.

pidProcess Specifies the process identifier of the process that receives access to the shared memory segment.

pselRecipient Points to the variable that receives the new segment selector.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosGiveSeg** function is successful even if the specified process already has access to the segment.

DosGiveSeg applies only to shared memory segments created by using the **DosAllocSeg** function with the *fAlloc* parameter set to SEG_GIVEABLE.

See Also

DosAllocSeg, DosGetSeg

DosHoldSignal

USHORT DosHoldSignal(*fDisable***)**
USHORT *fDisable*; /* disable/enable signals */

The **DosHoldSignal** function disables or enables signal processing for the current process.

The **DosHoldSignal** function is a family API function.

Parameters

fDisable Specifies whether to disable or enable signals that are intended for the current process. If this parameter is HLDSIG_DISABLE, the function disables signals. If it is HLDSIG_ENABLE, the function enables signals and restores the request count to its value before the last call to **DosHoldSignal**.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_FUNCTION

Comments

If the *fDisable* parameter is HLDSIG_DISABLE, the function holds all signals without processing them until **DosHoldSignal** is called again with *fDisable* set to HLDSIG_ENABLE. Signals should be held for as little time as possible; if necessary, a signal should be released and then held again.

Requests to disable and enable signal processing are cumulative. This means two requests to disable processing must be followed by two requests to enable processing before processing is enabled.

The **DosHoldSignal** function is intended to be used by library routines, subsystems, and similar code that need to prevent a possible signal from interfering with the completion of the current activity—for instance, activity in locked segments or in temporarily reserved resources.

Restrictions

In real mode, the following restriction applies to the **DosHoldSignal** function:

- Only the signal interrupt (SIG_CTRLC) and signal break (SIG_CTRLBREAK) signals are recognized.

Example

This example calls the **DosHoldSignal** function to disable signals and calls the **DosEnterCritSec** function to stop all other threads. When the processing of the critical section of code is completed, the **DosHoldSignal** function enables signals again:

```
DosHoldSignal(HLDSIG_DISABLE);        /* disables signals        */
DosEnterCritSec();                    /* enters critical section */
    .
    .
    .
DosExitCritSec();                     /* exits critical section  */
DosHoldSignal(HLDSIG_ENABLE);         /* enables signals         */
```

See Also

DosCLIAccess, DosEnterCritSec, DosFlagProcess

DosInsMessage

```
USHORT DosInsMessage(ppchVTable, usVCount, pszMsg, cbMsg, pchBuf, cbBuf, pcbMsg)
PCHAR FAR * ppchVTable;     /* pointer to table of character pointers  */
USHORT usVCount;            /* number of pointers in table             */
PSZ pszMsg;                 /* pointer to input message                */
USHORT cbMsg;               /* number of bytes in input message        */
PCHAR pchBuf;               /* pointer to buffer for updated message   */
USHORT cbBuf;               /* number of bytes in buffer               */
PUSHORT pcbMsg;             /* pointer to variable for length of message */
```

The **DosInsMessage** function copies a specified message to a buffer. Unlike the **DosGetMessage** function, **DosInsMessage** does not retrieve a message. **DosInsMessage** is often used when messages are loaded before the insertion-text strings are known.

The **DosInsMessage** function is a family API function.

Parameters

ppchVTable Points to a table of pointers to null-terminated strings than can be inserted into the message. Up to nine strings can be given.

usVCount Specifies the number of strings in the table. This parameter can be any value from 0 through 9. If this parameter is zero, the *ppchVTable* parameter is ignored. If this parameter is greater than 9, the function returns an error value indicating that the *usVCount* parameter is out of range.

pszMsg Points to a null-terminated string that specifies the message to process.

cbMsg Specifies the length (in bytes) of the message.

pchBuf Points to the buffer that receives the message.

cbBuf Specifies the length (in bytes) of the buffer that receives the message.

pcbMsg Points to the variable that receives the number of bytes copied to the buffer.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MR_INV_IVCOUNT
ERROR_MR_MSG_TOO_LONG

Comments

As it copies a message, the **DosInsMessage** function replaces any symbol in the form %*x* (where *x* is a digit from 1 through 9) with one of the strings pointed to in the table pointed to by the *ppchVTable* parameter. For example, the function replaces all symbols of the form %1 with the first string pointed to in the table. If there is no corresponding string in the table, **DosInsMessage** copies the %*x* sequence to the buffer. If the message is too long to fit in the buffer, the **DosGetMessage** function truncates the message and returns an error code.

Restrictions

In real mode, the following restriction applies to the **DosInsMessage** function:

- There is no method of identifying the boot drive. The system assumes that the message file is in the root directory of the current drive.

See Also

DosGetMessage, DosPutMessage

■ DosKillProcess

```
USHORT DosKillProcess(fScope, pidProcess)
USHORT fScope;       /* flag for process only-parent and child processes */
PID pidProcess;      /* process identifier of process to be ended        */
```

The **DosKillProcess** function terminates the specified process, with the option of also terminating all child processes that belong to it. Any subsequent request for the process's termination code returns the TC_KILLPROCESS code, unless the process intercepted the termination request.

Parameters

fScope Specifies whether to terminate the child processes that belong to the specified process that is terminated. If this parameter is DKP_PROCESSTREE, the function terminates the specified process and all of its child processes. If it is DKP_PROCESS, the function terminates the specified process only.

pidProcess Specifies the process identifier of the process to terminate.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_PROCID

Comments

A process can intercept the termination request generated by the **DosKill-Process** function by using the **DosSetSigHandler** function to create a signal handler. When the process creates a signal handler, the process typically completes any termination tasks, such as copying data from local buffers to files, then calls the **DosExit** function to terminate. If a process has no signal handler, the **DosKillProcess** function terminates the process after flushing all system file buffers and closing all handles opened by the process.

Before terminating, the process being terminated must flush file buffers that are not managed by MS OS/2—for example, the buffers managed by the C run-time library. MS OS/2 does not flush these buffers as part of its termination sequence.

Example

This example creates the child process *abc.exe*, then calls the **DosKillProcess** function to terminate it:

```
CHAR achFailName[128];
RESULTCODES resc;
DosExecPgm(achFailName, sizeof(achFailName),
    EXEC_ASYNCH, "abc ", 0, &resc, "abc.exe");
    .
    .
    .
DosKillProcess(DKP_PROCESS, resc.codeTerminate);
```

See Also

DosCwait, DosExit, DosSetSigHandler

DosLoadModule

```
USHORT DosLoadModule(pszFailName, cbFileName, pszModName, phmod)
PSZ pszFailName;          /* pointer to buffer for name if failure    */
USHORT cbFileName;        /* length of buffer for name if failure     */
PSZ pszModName;           /* pointer to module name                   */
PHMODULE phmod;           /* pointer to variable for module handle */
```

The **DosLoadModule** function loads a dynamic-link module and returns a handle for the module. You can use the module handle to retrieve the entry addresses of procedures in the module and to retrieve information about the module.

Parameters

pszFailName Points to the buffer that receives a null-terminated string. The **DosLoadModule** function copies a string to the buffer only if the function fails to load the module. The string identifies the dynamic-link module responsible for the failure. This module may be other than the one specified in the *pszModName* parameter if the specified module links to other dynamic-link modules.

cbFileName Specifies the length (in bytes) of the buffer pointed to by the *pszFailName* parameter.

pszModName Points to a null-terminated string. This string must be a valid MS OS/2 filename that specifies the path and filename of the dynamic-link module to be loaded. All dynamic-link modules have the *.dll* filename extension, by default.

phmod Points to the variable that receives the handle of the dynamic-link module.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_FORMAT
ERROR_FILE_NOT_FOUND
ERROR_INTERRUPT
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosLoadModule** function loads only MS OS/2 dynamic-link modules. Attempts to load other executable files (such as MS-DOS executable files) result in errors.

Example

This example calls the **DosLoadModule** function to load the dynamic-link module *qhdll.dll*. This example then calls the **DosGetProcAddr** function to retrieve the address of the BOXMESSAGE function that is defined in the module. After calling the BOXMESSAGE function, the example calls **DosFreeModule** to free the dynamic-link module. (This example is accurate if *qhdll.dll* exists in a directory defined by the **libpath** parameter of the *config.sys* file, and if *qhdll.dll* contains the BOXMESSAGE function that uses the Pascal calling convention.)

```
CHAR achFailName[128];
HMODULE hmod;
VOID (PASCAL FAR *pfnBoxMsg)(PSZ, BYTE, BYTE, SHANDLE, SHANDLE, BOOL);

DosLoadModule(achFailName,        /* failure name buffer         */
    sizeof(achFailName),          /* size of failure name buffer */
    "qhdll",                      /* module name                 */
    &hmod);                       /* address of handle           */
DosGetProcAddr(hmod, "BOXMESSAGE", &pfnBoxMsg);
pfnBoxMsg("Hello World", 0x30, 1, 0, 0, FALSE);
DosFreeModule(hmod);
```

See Also **DosExecPgm, DosFreeModule, DosGetModName, DosGetProcAddr**

DosLockSeg

USHORT DosLockSeg(*sel*)
SEL *sel*; /* selector of segment to lock */

The **DosLockSeg** function locks a discardable segment in memory. A locked segment cannot be discarded until it is unlocked by using the **DosUnlockSeg** function.

If a segment has been discarded, the **DosLockSeg** function returns an error value that specifies that the segment no longer exists. When this occurs, the **DosReallocSeg** function can be called to allocate a new copy of the segment. The program must recreate any discarded data.

Parameters

sel Specifies the selector of the segment to lock.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

DosLockSeg applies only to segments that have been allocated by using the **DosAllocSeg** function with the *fAlloc* parameter set to SEG_DISCARDABLE.

MS OS/2 can move and swap a locked segment as needed.

The **DosLockSeg** and **DosUnlockSeg** functions may be nested. For example, if **DosLockSeg** is called five times to lock a segment, **DosUnlockSeg** must be called five times to unlock the segment. A segment becomes permanently locked if it is locked 255 times without being unlocked.

See Also **DosAllocSeg, DosReallocSeg, DosUnlockSeg**

DosMakeNmPipe

```
USHORT DosMakeNmPipe(pszName, php, fsOpenMode, fsPipeMode, cbOutBuf, cbInBuf, ulTimeOut)
PSZ pszName;            /* pipe name                           */
PHPIPE php;             /* pointer to pipe handle              */
USHORT fsOpenMode;      /* open mode of pipe                   */
USHORT fsPipeMode;      /* pipe-specific modes                 */
USHORT cbOutBuf;        /* number of bytes in output buffer    */
USHORT cbInBuf;         /* number of bytes in input buffer     */
ULONG ulTimeOut;        /* timeout value                       */
```

The **DosMakeNmPipe** function creates a named pipe and retrieves a handle that can be used in subsequent pipe operations.

Parameters

pszName Points to a null-terminated string that identifies the name of the pipe. The string must have the following form:

\pipe*name*

The string name, *name*, must have the same format as an MS OS/2 filename.

php Points to the variable that receives the handle of the named pipe.

fsOpenMode Specifies the modes with which to open the pipe. This parameter is a combination of an access mode flag, an inheritance flag, and a write-behind flag. The possible values are:

Value	Meaning
PIPE_ACCESS_DUPLEX	Pipe is full duplex—going to and from server and client.
PIPE_ACCESS_INBOUND	Pipe is inbound—going from client to server.
PIPE_ACCESS_OUTBOUND	Pipe is outbound—going from server to client.
PIPE_INHERIT	Pipe is inherited by any child processes that are created by using the **DosExecPgm** function.
PIPE_NOINHERIT	Pipe is private to the current process and cannot be inherited.
PIPE_NOWRITEBEHIND	Write-behind to remote pipes is not allowed.
PIPE_WRITEBEHIND	Write-behind to remote pipes is allowed.

fsPipeMode Specifies the pipe-specific modes of the pipe. This parameter is a combination of an instance count, a read-mode flag, a type flag, and a wait flag. The possible values are:

Value	Meaning
PIPE_WAIT	Reading from and writing to the pipe waits if no data is available.
PIPE_NOWAIT	Reading from and writing to the pipe returns immediately if no data is available.

Value	Meaning
PIPE_READMODE_BYTE	Read pipe as a byte stream.
PIPE_READMODE_MESSAGE	Read pipe as a message stream.
PIPE_TYPE_BYTE	Pipe is a byte-stream pipe.
PIPE_TYPE_MESSAGE	Pipe is a message-stream pipe.
PIPE_UNLIMITED_INSTANCES	Unlimited instances of the pipe can be created. If this value is not specified, a value from 1 through 254 can be used for the number of instances.

cbOutBuf Specifies the number of bytes to reserve for the outgoing buffer.

cbInBuf Specifies the number of bytes to reserve for the incoming buffer.

ulTimeOut Specifies the default value (in milliseconds) of the timeout parameter of the **DosWaitNmPipe** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY
ERROR_OUT_OF_STRUCTURES
ERROR_PATH_NOT_FOUND
ERROR_PIPE_BUSY

See Also

DosClose, DosWaitNmPipe

■ DosMakePipe

USHORT DosMakePipe (*phfRead*, *phfWrite*, *cbPipe*)

PHFILE *phfRead*; /* pointer to variable for read handle */
PHFILE *phfWrite*; /* pointer to variable for write handle */
USHORT *cbPipe*; /* number of bytes reserved for pipe */

The **DosMakePipe** function creates a pipe. The function creates the pipe, assigning the specified pipe size to the storage buffer, and also creates handles that the process can use to read from and write to the buffer in subsequent calls to the **DosRead** and **DosWrite** functions.

Parameters

phfRead Points to the variable that receives the read handle for the pipe.

phfWrite Points to the variable that receives the write handle for the pipe.

cbPipe Specifies the size (in bytes) to allocate for the storage buffer for this pipe. This parameter can be any value up to 65,536 minus the size of the pipe header, which is currently 32 bytes. If this parameter is zero, the default buffer size is used.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_NOT_ENOUGH_MEMORY
ERROR_TOO_MANY_OPEN_FILES

Comments

Pipes are typically used by a pair of processes. One process creates the pipe and passes a handle to the other process. This lets one process write into the pipe and the other read from the pipe. Since MS OS/2 provides no permission checks on pipes, the cooperating processes must ensure that they do not attempt to write to or read from the pipe at the same time.

When all of a pipe's handles are closed by using the **DosClose** function, MS OS/2 deletes that pipe. If two processes are communicating by using a pipe and the process ends that is reading the pipe, the next call to the **DosWrite** function for that pipe returns the "broken pipe" error value.

MS OS/2 temporarily blocks any call to the **DosWrite** function that would have written more data to the pipe than could fit in the storage buffer. The system removes the block as soon as enough data is read from the pipe to make room for the remaining unwritten data.

See Also

DosClose, DosDupHandle, DosRead, DosWrite

■ DosMemAvail

USHORT DosMemAvail(*pulAvailMem***)**
PULONG *pulAvailMem*; /* pointer to variable for available memory */

The **DosMemAvail** function retrieves the size of the largest block of free memory available when the function is called. The largest free block consists of all free memory, whether consecutive or not. This function does not cause segments to be moved, swapped, or discarded.

Parameters

pulAvailMem Points to the variable that receives the size (in bytes) of the largest free block of memory.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

Since other processes may allocate and free memory at any time, the size of the largest free block can be expected to change.

The **DosMemAvail** function returns only the amount of memory currently available without swapping. More memory can be allocated than indicated by the **DosMemAvail** function—when necessary, the system swaps memory or discards unlocked memory to meet memory-allocation requests.

Example

This example calls **DosMemAvail** to determine the amount of available memory. It then allocates one third of that memory and allows for reallocation of up to ten 64K segments.

```
#define SEGSIZE (64L * 1024L)
LONG lAvail;
SEL sel;

DosMemAvail(&lAvail);                       /* gets amount of current memory */
lAvail /= 3L;                               /* calculate one third of memory */
DosAllocHuge((USHORT) (lAvail / SEGSIZE), /* number of segments         */
    (USHORT) (lAvail % SEGSIZE),          /* size of last segment       */
    &sel,                                 /* address of selector        */
    10,                                 /* allows reallocation up to 640K */
    SEG_NONSHARED);                     /* sharing flag                   */
```

See Also **DosAllocHuge**

DosMkDir

USHORT DosMkDir(*pszDirName*, *ulReserved*)
PSZ *pszDirName*; /* new directory name */
ULONG *ulReserved*; /* must be zero */

The **DosMkDir** function creates the specified directory. If the directory already exists or the specified directory name is invalid, the function returns an error value.

The **DosMkDir** function is a family API function.

Parameters

pszDirName Points to a null-terminated string. This string must be a valid MS OS/2 directory name.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_DRIVE_LOCKED
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND

Example

This example calls the **DosMkDir** function to create the subdirectory *abc* and report an error if it fails:

```
USHORT usError;
usError = DosMkDir("abc", 0L);
if (usError)
    VioWrtTTY("Can't open directory\r\n", 22, 0);
else {
```

See Also **DosRmDir**

DosMonClose

USHORT DosMonClose(*hmon*)
HMONITOR *hmon*; /* monitor handle to close */

The **DosMonClose** function closes the specified monitor. The function flushes and closes all monitor buffers associated with this process.

Parameters

hmon Identifies the monitor to close. This handle must have been created previously by using the **DosMonOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MON_INVALID_HANDLE

See Also

DosMonOpen, DosMonRead, DosMonReg, DosMonWrite

DosMonOpen

```
USHORT DosMonOpen(pszDevName, phmon)
PSZ pszDevName;          /* pointer to device name              */
PHMONITOR phmon;         /* pointer to variable for monitor handle */
```

The **DosMonOpen** function opens a monitor and creates a handle that can be used to identify the monitor. Only one monitor per process is allowed—that is, **DosMonOpen** must not be called more than once by any process.

Parameters

pszDevName Points to a null-terminated string. This string specifies the name of the device for which the monitor is to be opened.

phmon Points to the variable that receives the monitor handle.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MON_INVALID_DEVNAME
ERROR_NOT_ENOUGH_MEMORY

Comments

You can determine whether a device supports a monitor by using the **DosDevIOCtl** function. For more information, see DEV_QUERYMONSUPPORT in Chapter 3, "Input-and-Output Control Functions."

See Also

DosMonClose, DosMonRead, DosMonWrite

DosMonRead

```
USHORT DosMonRead(pbInBuffer, fWait, pbDataBuf, pcbDataBuf)
PBYTE pbInBuffer;        /* pointer to buffer for monitor input     */
UCHAR fWait;             /* wait/no-wait flag                       */
PBYTE pbDataBuf;         /* pointer to buffer for data records      */
PUSHORT pcbDataBuf;      /* pointer to variable with size of buffer */
```

The **DosMonRead** function reads data records from the device associated with the specified monitor and copies the records to a buffer.

Parameters

pbInBuffer Points to the buffer for monitor input. This handle must have been registered previously by using the **DosMonReg** function.

fWait Specifies whether the function should wait for input. If this parameter is DCWW_WAIT, the function waits until input is ready. If this parameter is DCWW_NOWAIT, no input is ready, and the function returns immediately.

pbDataBuf Points to the buffer that receives the data records.

pcbDataBuf Points to the variable that contains the size (in bytes) of the buffer that receives the data records. When the **DosMonRead** function returns, it sets the variable to the number of bytes copied from the data record to the buffer.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MON_BUFFER_EMPTY
ERROR_MON_BUFFER_TOO_SMALL
ERROR_MON_INVALID_PARMS

Comments

Device monitors must respond rapidly to avoid delaying input and output (I/O). (This rapid response is especially important for keyboard monitors.) A monitor process should be written so that any threads that read and write the monitor data run at a high priority. These threads should never perform operations that might delay them, such as waiting for I/O or a semaphore. The monitor process can have other threads running at normal priority to handle such operations.

See Also

DosMonClose, DosMonOpen, DosMonReg, DosMonWrite

DosMonReg

```
USHORT DosMonReg(hmon, pbInBuf, pbOutBuf, fPosition, usIndex)
HMONITOR hmon;      /* monitor handle to register             */
PBYTE pbInBuf;      /* pointer to structure for input buffer  */
PBYTE pbOutBuf;     /* pointer to structure for output buffer */
USHORT fPosition;   /* position flag                          */
USHORT usIndex;     /* index                                  */
```

The **DosMonReg** function registers a monitor by placing it in a chain of other monitors for the same device. Each monitor receives input from or sends output to the device in the order in which it appears in the chain.

Parameters

hmon Identifies the monitor to register. This handle must have been created previously by using the **DosMonOpen** function.

pbInBuf Points to the **MONIN** structure that receives data from the device driver or from the previous monitor in the chain. The **MONIN** structure has the following form:

```
typedef struct _MONIN {
    USHORT cb;
    BYTE abReserved[18];
    BYTE bBuffer[108];
} MONIN;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbOutBuf Points to the **MONOUT** structure that receives data for the next monitor in the chain. The **MONOUT** structure has the following form:

```
typedef struct _MONOUT {
    USHORT cb;
    BYTE abReserved[18];
    BYTE abBuffer[108];
} MONOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

fPosition Specifies the position of the monitor in the chain of input and output. This parameter can be one of the following values:

Value	Meaning
MONITOR_BEGIN	Place the monitor at the beginning of the chain, in front of any other monitors already in the chain.
MONITOR_DEFAULT	Place the monitor anywhere in the chain.
MONITOR_END	Place the monitor at the end of the chain, after any other monitors already in the chain.

usIndex Specifies a device-specific value. If the device is the keyboard, this parameter specifies the identifier for the screen group to monitor. If no screen-group number is available (because the monitor is detached), the identifier of the current foreground screen group can be obtained by calling the **DosGetInfoSeg** function. (The current foreground screen group is the screen group that made the most recent call to the **KbdCharIn** function.)

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MON_BUFFER_TOO_SMALL
ERROR_MON_INVALID_HANDLE
ERROR_MON_INVALID_PARMS
ERROR_NOT_ENOUGH_MEMORY

Comments

The **MONIN** and **MONOUT** structures must be in the same segment.

See Also

DosMonClose, DosMonOpen, DosMonRead, DosMonWrite, KbdCharIn

DosMonWrite

```
USHORT DosMonWrite(pbOutBuf, pbDataBuf, cbDataBuf)
PBYTE pbOutBuf;       /* monitor-output buffer              */
PBYTE pbDataBuf;      /* buffer from which records are taken */
USHORT cbDataBuf;     /* number of bytes                    */
```

The **DosMonWrite** function writes one or more data records into a device's output stream. The output-buffer structure identifies the device that receives the data records.

Parameters

pbOutBuf Points to the output-buffer structure for the monitor. The monitor must have been registered previously by using the **DosMonReg** function.

pbDataBuf Points to the buffer that contains the data records to insert into the device's output stream.

cbDataBuf Specifies the number of bytes of data records in the buffer pointed to by the *pbDataBuf* parameter.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MON_DATA_TOO_LARGE
ERROR_MON_INVALID_PARMS

Comments

Device monitors must respond rapidly to avoid delaying input and output (I/O). (This rapid response is especially important for keyboard monitors.) A monitor process should be written so that any threads that read and write the monitor data run at a high priority. These threads should never perform operations that might delay them, such as waiting for I/O or a semaphore. The monitor process can have other threads running at normal priority to handle such operations.

See Also

DosMonClose, DosMonOpen, DosMonRead, DosMonReg

■ DosMove

```
USHORT DosMove(pszOldName, pszNewName, ulReserved)
PSZ pszOldName;      /* pointer to old path and filename */
PSZ pszNewName;      /* pointer to new path and filename */
ULONG ulReserved;    /* must be zero                     */
```

The **DosMove** function moves a specified file to a specified new directory and/or filename. The function is often used to rename an existing file by moving the file to a new filename location in the same directory. The function can also be used to move a file to a new directory while preserving the existing filename or to rename any directory that is not the root directory.

The **DosMove** function is a family API function.

Parameters

pszOldName Points to a null-terminated string. This string specifies the current filename of the file to be moved. The string must be a valid MS OS/2 filename.

pszNewName Points to a null-terminated string. This string specifies the new directory and filename of the file to be moved. The string must be a valid MS OS/2 filename.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_NOT_DOS_DISK
ERROR_NOT_SAME_DEVICE
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION

Comments

The **DosMove** function cannot move a file from one drive to another; if a drive is used in the *pszOldName* string, the same drive must be used in the *pszNewName* string.

Wildcard characters are not allowed in the filename.

Example

This example calls the **DosMove** function to move the file *abc* to the root directory of the current drive and to rename the file *xyz*. This does not copy the file, but it may change the subdirectory that the filename appears in and may change the filename itself.

```
DosMove("abc",              /* old filename and path */
    "\\xyz",                /* new filename and path */
    0L);                    /* reserved              */
```

See Also

DosDelete, DosSelectDisk

DosMuxSemWait

```
USHORT DosMuxSemWait(pisemCleared, pmsxl, lTimeOut)
PUSHORT pisemCleared;   /* pointer to variable for cleared semaphore     */
PVOID pmsxl;            /* pointer to structure containing semaphore list */
LONG lTimeOut;          /* time-out value                                */
```

The **DosMuxSemWait** function waits for one or more of the specified semaphores to clear. The function first checks the semaphores specified in the list pointed to by the *pmsxl* parameter. If any of the semaphores in this list are clear, the function returns. Otherwise, the function waits until the time specified by the *lTimeOut* parameter elapses or until one of the semaphores in the list clears.

The semaphore list can contain up to 16 semaphores.

Parameters

pisemCleared Points to the variable that receives the index number of the most recently cleared semaphore.

pmsxl Points to the **MUXSEMLIST** structure containing a semaphore list that defines the semaphores to be cleared. The semaphore list consists of one or more semaphore handles. The **MUXSEMLIST** structure has the following form:

```
typedef struct _MUXSEMLIST {
    USHORT cmxs;
    MUXSEM amxs[16];
} MUXSEMLIST;
```

The structure may contain up to 16 semaphores.

For a full description, see Chapter 4, "Types, Macros, Structures."

lTimeOut Specifies how long to wait for the semaphores to become available. If the value is greater then zero, this parameter specifies the number of milliseconds to wait before returning. If it is SEM_IMMEDIATE_RETURN, the function returns immediately. If it is SEM_INDEFINITE_WAIT, the function waits indefinitely.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_EXCL_SEM_ALREADY_OWNED
ERROR_INTERRUPT
ERROR_INVALID_EVENT_COUNT
ERROR_INVALID_HANDLE
ERROR_INVALID_LIST_FORMAT
ERROR_SEM_TIMEOUT
ERROR_TOO_MANY_MUXWAITERS

Comments

Although it is declared with the **PVOID** type, the second parameter of the **DosMuxSemWait** function must point to a **MUXSEMLIST** structure. You can create the structure by using the **DEFINEMUXSEMLIST** macro. The macro has the following syntax:

DEFINEMUXSEMLIST(*name, size*)

The *name* parameter specifies the name of the structure to be created, and the *size* parameter specifies the number of elements in the structure—that is, the number of semaphores in the list. This macro creates an array of **MUXSEMLIST** structures.

Unlike the other blocking semaphore functions (**DosSemRequest**, **DosSemSetWait** and **DosSemWait**), **DosMuxSemWait** returns whenever one of the semaphores on its list is cleared, regardless of how long that semaphore may remain cleared. It is possible that the semaphore could be reset before the **DosMuxSemWait** function returns.

The **DosMuxSemWait** function does not set or claim any of the semaphores.

The **DosMuxSemWait** function can be used in conjunction with one or more semaphores as a triggering or synchronizing device. One or more threads can use **DosMuxSemWait** to wait for a semaphore. When an event occurs, another thread can clear that semaphore and immediately set it again. Any threads that waited for that semaphore by using **DosMuxSemWait** will return. Threads that were waiting by using one of the "level-triggered" functions (**DosSemRequest**, **DosSemSetWait**, or **DosSemWait**) may or may not resume, depending on the scheduler's dispatch order and the activity of other threads in the system.

Example

This example creates a structure of system semaphore handles for use by the **DosMuxSemWait** function. It sets the first element of the structure to the number of handles stored and creates two semaphore handles. It then calls **DosMuxSemWait** to wait until one of the semaphores is cleared. It uses the value of the *usSemIndex* parameter to find out which semaphore is cleared, and if it is semaphore 1, the example sets that semaphore.

```
DEFINEMUXSEMLIST(MuxList, 2)        /* creates structure array */
USHORT usSemIndex;
MuxList.cmxs = 2;
DosCreateSem(CSEM_PUBLIC, &MuxList.amxs[0].hsem,
    "\\sem\\timer0.sem");
DosCreateSem(CSEM_PUBLIC, &MuxList.amxs[1].hsem,
    "\\sem\\timer1.sem");
    .
    .
    .
DosMuxSemWait(&usSemIndex, &MuxList, 5000L);
if (usSemIndex == 1) {
    DosSemSet(MuxList.amxs[1].hsem);
```

See Also **DosCreateSem, DosSemRequest, DosSemSet, DosSemSetWait, DosSemWait, WinMsgMuxSemWait**

DosNewSize

USHORT DosNewSize(*hf*, *ulNewSize*)
HFILE *hf*; /* file handle */
ULONG *ulNewSize*; /* new size of file */

The **DosNewSize** function changes the size of the specified file. The function can be used to truncate or extend a file. If a file is extended, the value of the new bytes is undefined.

The **DosNewSize** function is a family API function.

Parameters

hf Identifies the file to be changed. This handle must have been created previously by using the **DosOpen** function.

ulNewSize Specifies the file's new size (in bytes).

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_DISK_FULL
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_LOCK_VIOLATION
ERROR_NOT_DOS_DISK

Comments

The **DosNewSize** function applies only to files that have been opened for writing. To change the size of a read-only file, first change the file's attributes by using the **DosSetFileMode** function, then open the file for writing.

If the function extends a file, the system will attempt to allocate sectors that are contiguous with the existing file sectors.

Example

This example opens the file *abc* and calls the **DosNewSize** function to set the file's size to 100 bytes. If the file already exists and is larger than 100 bytes, it is truncated to 100 bytes. If the file is smaller than 100 bytes, or if it was created by using the **DosOpen** function, it is expanded to 100 bytes.

```
HFILE hf;
USHORT usAction;
DosOpen("abc", &hf, &usAction, OL, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE,
    OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYREADWRITE, OL);
DosNewSize(hf, 100L);
```

See Also **DosOpen, DosQFileInfo, DosSetFileMode**

DosOpen

USHORT DosOpen(*pszFileName*, *phf*, *pusAction*, *ulFileSize*, *usAttribute*, *fsOpenFlags*, *fsOpenMode*, *ulReserved***)**

```
PSZ pszFileName;        /* pointer to filename                        */
PHFILE phf;             /* pointer to variable for file handle        */
PUSHORT pusAction;      /* pointer to variable for action taken       */
ULONG ulFileSize;       /* file size if created or truncated          */
USHORT usAttribute;     /* file attribute                             */
USHORT fsOpenFlags;     /* action taken if file exists/does not exist */
USHORT fsOpenMode;      /* open mode of file                          */
ULONG ulReserved;       /* must be zero                               */
```

The **DosOpen** function opens an existing file or creates a new file. This function returns a handle that can be used to read from and write to the file, as well as to retrieve information about the file.

The **DosOpen** function is a family API function.

Parameters

pszFileName Points to the null-terminated string that specifies the name of the file to be opened. The string must be a valid MS OS/2 filename and must not contain wildcard characters.

phf Points to the variable that receives the handle of the opened file.

pusAction Points to the variable receiving the value that specifies the action taken by the **DosOpen** function. If **DosOpen** fails, this value has no meaning. Otherwise, it is one of the following values:

Value	Meaning
FILE_CREATED	File was created.
FILE_EXISTED	File already existed.
FILE_TRUNCATED	File existed and was truncated.

ulFileSize Specifies the file's new size (in bytes). This parameter applies only if the file is created or truncated. The size specification has no effect on a file that is opened only for reading.

usAttribute Specifies the file attributes. This parameter can be a combination of the following values:

Value	Meaning
FILE_NORMAL	File can be read from or written to.
FILE_READONLY	File can be read from, but not written to.

Value	Meaning
FILE_HIDDEN	File is hidden and does not appear in a directory listing.
FILE_SYSTEM	File is a system file.
FILE_ARCHIVED	File has been archived.

File attributes apply only if the file is created.

fsOpenFlags Specifies the action to take both when the file exists and when it does not exist. This parameter may be one of the following values:

Value	Meaning
FILE_CREATE	Create a new file; fail if the file already exists.
FILE_OPEN	Open an existing file; fail if the file does not exist.
FILE_OPEN | FILE_CREATE	Open an existing file or create the file if it does not exist.
FILE_TRUNCATE	Open an existing file and change to a given size.
FILE_TRUNCATE | FILE_CREATE	Open an existing file and truncate it, or create the file if it does not exist.

fsOpenMode Specifies the modes with which to open the file. It consists of one access mode and one share mode. The other values are option and can be given in any combination:

Value	Meaning
OPEN_ACCESS_READONLY	Data may be read from the file but not written to it.
OPEN_ACCESS_READWRITE	Data may be read from or written to the file.
OPEN_ACCESS_WRITEONLY	Data may be written to the file but not read from it.
OPEN_SHARE_DENYNONE	Other processes can open the file for any access: read-only, write-only, or read-write.
OPEN_SHARE_DENYREAD	Other processes can open the file for write-only access but they cannot open it for read-only or read-write access.
OPEN_SHARE_DENYREADWRITE	The current process has exclusive access to the file. The file cannot be opened by any process (including the current process).

Value	Meaning
OPEN_SHARE_DENYWRITE	Other processes can open the file for read-only access but they cannot open it for write-only or read-write access.
OPEN_FLAGS_DASD	The file handle represents a physical drive that has been opened for direct access. (The *pszFileName* parameter must specify a drive name.) The **DosDevIOCtl** function can be used with this file handle to bypass the file system and to access the sectors of the drive directly.
OPEN_FLAGS_FAIL_ON_ERROR	Any function that uses the file handle returns immediately with an error value if there is an I/O error—for example, when the drive door is open or a sector is missing. If this value is not specified, the system passes the error to the system critical-error handler, which then reports the error to the user with a hard-error popup. The fail-on-error flag is not inherited by child processes. The fail-on-error flag applies to all functions that use the file handle, with the exception of the **DosDevIOCtl** function.
OPEN_FLAGS_NOINHERIT	The file handle is not available to any child process started by the current process. If this value is not specified, any child process started by the current process may use the file handle.
OPEN_FLAGS_WRITE_THROUGH	This flag applies to functions, such as **DosWrite**, that write data to the file. If this value is specified, the system writes data to the device before the given function returns. Otherwise, the system may store the data in an internal file buffer and write the data to the device only when the buffer is full or the file is closed.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_CANNOT_MAKE
ERROR_DISK_FULL
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_INVALID_ACCESS
ERROR_INVALID_PARAMETER
ERROR_NOT_DOS_DISK
ERROR_OPEN_FAILED
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION
ERROR_TOO_MANY_OPEN_FILES

Comments

The ERROR_ACCESS_DENIED value is returned if you try to open a file in a mode that is incompatible with the file's current access and sharing modes—for example, if you attempt to open a read-only file for writing. This error is also returned if some other process has opened the file with a sharing method that denies the type of access you have requested.

Once the file is opened, the **DosSetFHandState** function can be used to change the OPEN_FLAGS_FAIL_ON_ERROR, OPEN_FLAGS_NOINHERIT, and OPEN_FLAGS_WRITE_THROUGH flags specified in the *fsOpenMode* parameter.

MS OS/2 does not provide a built-in method to inform a child process that it has inherited a given file handle. The parent process must pass this information to a child process. If the file is created without the OPEN_FLAGS_NOINHERIT flag, and the parent process terminates without closing the file, the file will remain open until all child processes have terminated.

Restrictions

In real mode, the following restriction applies to the **DosOpen** function:

- Only the access modes and the OPEN_FLAGS_DASD flag may be specified for the *fsOpenMode* parameter.

Example

This example calls the **DosOpen** function to create a file *abc* that is 100 bytes long and open it for write-only access. The *fsOpenFlags* parameter is set to FILE_CREATE so that **DosOpen** will return an error if the file already exists.

```
HFILE hf;
USHORT usAction;
DosOpen("abc",                                  /* filename to open       */
    &hf,                                        /* address of file handle */
    &usAction,                                  /* action taken           */
    100L,                                       /* size of new file       */
    FILE_NORMAL,                                /* file attribute         */
    FILE_CREATE,                                /* create the file        */
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, /* open mode              */
    0L);                                        /* reserved               */
```

See Also

DosBufReset, **DosChgFilePtr**, **DosDevIOCtl**, **DosDupHandle**, **DosExecPgm**, **DosQFHandState**, **DosQFileInfo**, **DosQFileMode**, **DosQFSInfo**, **DosSetFHandState**, **DosSetFileMode**

■ DosOpenQueue

```
USHORT DosOpenQueue(ppidOwner, phqueue, pszQueueName)
PUSHORT ppidOwner;      /* pointer to variable for queue owner's identifier */
PHQUEUE phqueue;        /* pointer to variable for handle of queue          */
PSZ pszQueueName;       /* pointer to name of queue                         */
```

The **DosOpenQueue** function opens a queue for the current process.

Parameters

ppidOwner Points to the variable that receives the process identifier of the queue owner.

phqueue Points to the variable that receives the handle of the queue.

pszQueueName Points to a null-terminated string. This string identifies the queue and must have the following form:

\queues*name*

The string name, *name*, must have the same format as an MS OS/2 filename and must identify a queue that has been created previously by using the **DosCreateQueue** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_QUE_NAME_NOT_EXIST
ERROR_QUE_NO_MEMORY

See Also

DosCloseQueue, DosCreateQueue, DosReadQueue, DosWriteQueue

■ DosOpenSem

```
USHORT DosOpenSem(phsem, pszSemName)
PHSEM phsem;        /* pointer to variable for semaphore handle */
PSZ pszSemName;     /* pointer to semaphore name                */
```

The **DosOpenSem** function opens a system semaphore of the specified name and returns a unique semaphore handle. The semaphore handle can then be used to set and clear the semaphore and to carry out other tasks that use the semaphore.

Parameters

phsem Points to the variable that receives the new semaphore handle.

pszSemName Points to the null-terminated string that identifies the semaphore. The string must have the following form:

\sem*name*

The string name, *name*, must have the same format as an MS OS/2 filename and must identify a semaphore that has been created previously by using the **DosCreateSem** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_NAME
ERROR_SEM_NOT_FOUND
ERROR_TOO_MANY_SEMAPHORES

Comments

The **DosOpenSem** function only returns the handle of the semaphore; it does not test or change the value of the semaphore. The semaphore handle is the same as the semaphore handle returned by the **DosCreateSem** function that created the semaphore.

If a process creates a child process by using the **DosExecPgm** function, the new process inherits any open semaphore handles.

Under MS OS/2, system semaphores reside in a memory buffer rather than a disk file. When the last process with an open semaphore terminates, that semaphore is closed and is no longer available to any other process.

Example

This example calls the **DosOpenSem** function to open a system semaphore that had been created previously:

```
HSEM hsem;                              /* handle to semaphore   */
DosOpenSem(&hsem, "\\sem\\abc.ext");    /* opens the semaphore   */
     .
     .
     .
DosCloseSem(hsem);                      /* closes the semaphore */
```

See Also

DosCloseSem, DosCreateSem, DosExecPgm, DosSemClear, DosSemRequest

DosPeekNmPipe

```
USHORT DosPeekNmPipe(hp, pbBuf, cbBuf, pcbRead, pcbAvail, pfsState)
HPIPE hp;              /* pipe handle                                */
PBYTE pbBuf;           /* pointer to buffer for data                 */
USHORT cbBuf;          /* length of buffer for data                  */
PUSHORT pcbRead;       /* pointer to variable for number bytes read  */
PUSHORT pcbAvail;      /* pointer to variable for number bytes available */
PUSHORT pfsState;      /* pointer to variable for pipe state         */
```

The **DosPeekNmPipe** function copies a pipe's data into a buffer.

Parameters

hp Identifies the pipe to read from.

pbBuf Points to a buffer that receives the data from the pipe.

cbBuf Specifies the length (in bytes) of the buffer that receives the data from the pipe.

pcbRead Points to the variable that receives a value specifying the number of bytes read from the pipe.

pcbAvail Points to the variable that the receives a value specifying the number of bytes that were available to be read. The first two bytes of this buffer specify the number of bytes remaining in the pipe (including message-header bytes). The next two bytes specify the number of bytes remaining in the current message. (There will be zero bytes remaining in the current message for a byte-stream pipe.)

pfsState Points to the variable that receives the state of the pipe. The state may be one of the following values:

Value	Meaning
PIPE_STATE_CLOSING	The pipe is closed and can no longer be used.
PIPE_STATE_CONNECTED	The pipe has been opened and is available for reading and writing.
PIPE_STATE_DISCONNECTED	The serving end must call the **DosConnectNmPipe** function to put the pipe into a listening state before a call to the **DosOpen** function will be accepted. A pipe is in a disconnected state between a call to the **DosMakeNmPipe** function and a call to the **DosConnectNmPipe** function.
PIPE_STATE_LISTENING	The pipe will accept a call to the **DosOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_PIPE_NOT_CONNECTED

Comments

The **DosPeekNmPipe** function never blocks, regardless of the blocking mode of the pipe.

If the **DosDisConnectNmPipe** function has been called, the pipe will remain disconnected until a call is made to the **DosConnectNmPipe** function.

See Also

DosConnectNmPipe, DosDisConnectNmPipe, DosMakeNmPipe, DosRead

DosPeekQueue

```
USHORT DosPeekQueue(hqueue, pqresc, pcbElement, ppv, pusElementCode, fWait, pbElemPrty, hsem)
HQUEUE hqueue;                  /* handle of queue to read from                   */
PQUEUERESULT pqresc;            /* pointer to structure for PID and request code */
PUSHORT pcbElement;             /* pointer to variable for number of bytes       */
PVOID FAR * ppv;                /* pointer to buffer for element received        */
PUSHORT pusElementCode;         /* pointer to variable for element position      */
UCHAR fWait;                    /* wait/no wait indicator                         */
PBYTE pbElemPrty;               /* pointer to variable for priority of element   */
ULONG hsem;                     /* semaphore handle                               */
```

The **DosPeekQueue** function retrieves an element without removing it from a queue. It copies the address of the element to a pointer and fills a structure with information about the element.

Parameters

hqueue Identifies the queue to be read from. This handle must have been previously created or opened by using the **DosCreateQueue** or **DosOpenQueue** function.

pqresc Points to the structure that receives information about the request. The **QUEUERESULT** structure has the following form:

```
typedef struct _QUEUERESULT {
    PID pidProcess;
    USHORT usEventCode;
} QUEUERESULT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pcbElement Points to the variable that receives the length in bytes of the element.

ppv Points to a pointer that receives the address of the element in the queue.

pusElementCode Points to the variable that specifies where to look in the queue for the element. If the *pusElementCode* parameter is 0x0000, the function looks at the beginning of the queue. Otherwise, the function assumes the value is an element identifier and looks for the element that immediately follows the specified element. When the function returns, it copies the identifier of the retrieved element to the variable. The element identifier can then be used to search for the next element or to read the given element from the queue.

fWait Specifies whether the function should wait for an element to be placed in the queue, if the queue is empty. If the *fWait* parameter is DCWW_WAIT, the function waits until an element is available. If it is DCWW_NOWAIT, the function returns immediately.

pbElemPrty Points to a variable that receives the priority value specified when the element was added to the queue. This is a numeric value from 0 through 15; 15 is the highest priority.

hsem Identifies a semaphore. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore. This semaphore would typically be used in a call to the **DosMuxSemWait** function to wait until the queue has an element. If the *fWait* parameter is DCWW_WAIT, *hsem* is ignored.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_QUE_ELEMENT_NOT_EXIST
ERROR_QUE_EMPTY
ERROR_QUE_INVALID_HANDLE
ERROR_QUE_INVALID_WAIT
ERROR_QUE_PROC_NOT_OWNED

Comments

If the queue is empty, the **DosPeekQueue** function either returns immediately or waits for an element to be written to the queue, depending on the value of the *fWait* parameter.

Only the process that created the queue may call the **DosPeekQueue** function.

See Also

DosCreateQueue, DosCreateSem, DosMuxSemWait, DosOpenSem, DosOpenQueue, DosReadQueue

■ DosPhysicalDisk

```
USHORT DosPhysicalDisk(usFunction, pbOutBuf, cbOutBuf, pbParmBuf, cbParmBuf)
USHORT usFunction;    /* action to take                    */
PBYTE pbOutBuf;       /* pointer to output buffer          */
USHORT cbOutBuf;      /* output-buffer length              */
PBYTE pbParmBuf;      /* pointer to user-supplied information */
USHORT cbParmBuf;     /* length of user-supplied information  */
```

The **DosPhysicalDisk** function retrieves information about partitionable disks.

Parameters

usFunction Specifies the action to take. It can be one of the following values:

Value	Meaning
INFO_COUNT_PARTITIONABLE_DISKS	Retrieve the total number of partitionable disks.
INFO_FREEIOCTLHANDLE	Release the handle obtained by a previous call to **DosPhysicalDisk**.
INFO_GETIOCTLHANDLE	Retrieve a handle to use with Category 9 **IOCtl** functions.

pbOutBuf Points to the buffer that receives output information. For a full description, see the first list under "Comments."

cbOutBuf Specifies the length (in bytes) of the output buffer.

pbParmBuf Points to a buffer that contains parameter data. For a full description, see the second list under "Comments."

cbParmBuf Specifies the length (in bytes) of the parameter buffer.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

When **DosPhysicalDisk** is used to obtain a handle to a partitionable physical drive (a *usFunction* value of INFO_GETIOCTLHANDLE), the *pbParmBuf* parameter should point to a null-terminated string that contains the drive number and a colon (:). The *cbParmBuf* parameter must contain the length of the entire string, including the trailing null character. For example, to obtain a handle for the first partitionable disk, *pbParmBuf* should point to "1:" and *cbParmBuf* should be 3.

The organization and content of the output buffer depend on the given function, as follows:

Function	cbOutBuf	Returned information
1	2	Total number of partitionable disks in system (one-based).
2	2	Handle for the specified partitionable disk for the Category 9 **IOCtl** functions.
3	0	None. Pointer must be zero.

This organization and content of the parameter buffer depend on the given function, as follows:

Function	cbParmBuf	Input parameters
1	0	None. Must be zero.
2	Length of string, including terminal null character	Null-terminated string that specifies the partitionable disk. The string must have the following form: *number*: The *number* parameter specifies the partitionable disk number. Partitionable disk numbers start at 1.
3	2	Handle retrieved by function 2.

Example

This example calls the **DosPhysicalDisk** function to determine the total number of partitionable disks. The total value is placed in the *usDataBuffer* variable.

```
USHORT usDataBuffer;
DosPhysicalDisk(INFO_COUNT_PARTITIONABLE_DISKS,
    (PBYTE) &usDataBuffer,      /* address of data buffer    */
    2,                          /* length of data buffer     */
    NULL,                       /* pointer to parameter list */
    0);                         /* length of parameter list  */
```

See Also

DosDevConfig, DosDevIOCtl

DosPortAccess

USHORT DosPortAccess(*usReserved***,** *fRelease***,** *usFirstPort***,** *usLastPort***)**

```
USHORT usReserved;    /* must be zero                */
USHORT fRelease;      /* request/release indicator   */
USHORT usFirstPort;   /* first port number           */
USHORT usLastPort;    /* last port number            */
```

The **DosPortAccess** function requests or releases access to a port, or ports, for input/output privilege.

Parameters

usReserved Specifies a reserved value; must be zero.

fRelease Specifies the type of access request. If this parameter is FALSE, the function requests access to a port. If it is TRUE, the function releases access to a port.

usFirstPort Specifies either a single port or the starting port number (start-of-range) in a contiguous range.

usLastPort Specifies either a single port or the ending port number (end-of-range) in a contiguous range. If only one port is being used, the *usFirstPort* and *usLastPort* parameters must be the same.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

Comments Programs that perform input or output (I/O) to a port, or ports, in IOPL segments must request port access from the operating system.

Granting port access automatically grants **cli** and **sti** privileges from the operating system. Therefore, there is no need to make an additional call to the **DosCLIAccess** function.

See Also **DosCLIAccess**

■ DosPTrace

USHORT DosPTrace (*pvPtraceBuf*)
PVOID *pvPtraceBuf*; /* pointer to structure receiving register values */

The **DosPTrace** function provides access to the MS OS/2 debugging functions. These debugging functions are available to any process that starts a protected-mode child process by using the **DosExecPgm** function with the *fExecFlags* parameter set to EXEC_TRACE.

Parameters *pvPtraceBuf* Points to the **PTRACEBUF** structure that receives the current values of the child process's registers and a code that indicates the reason for returning. The **PTRACEBUF** structure has the following form:

```
typedef struct PTRACEBUF {
    PID    pid;
    TID    tid;
    USHORT cmd;
    USHORT value;
    USHORT offv;
    USHORT segv;
    USHORT mte;
    USHORT rAX;
    USHORT rBX;
    USHORT rCX;
    USHORT rDX;
    USHORT rSI;
    USHORT rDI;
    USHORT rBP;
    USHORT rDS;
    USHORT rES;
    USHORT rIP;
    USHORT rCS;
    USHORT rF;
    USHORT rSP;
    USHORT rSS;
} PTRACEBUF;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_INVALID_FUNCTION
ERROR_INVALID_PROCID

Comments

To use the **DosPTrace** function, you need to provide the following function prototype in your source file:

USHORT DosPTrace(PVOID);

The **DosPTrace** function lets a parent process control the execution of the child process and access the child process's memory directly to insert break points or change data.

The parent process starts the child process to be debugged, then stops the child process by using the **DosPTrace** function with the **cmd** field of the **PTRACEBUF** structure set to 0x000A. The parent process can then insert break points or change memory in the child process by using **DosPTrace** and the **cmd** field values. Next, the parent process can start execution by setting the **cmd** field to 0x0007 (go until break point) or 0x0009 (single step). The parent process can set initial register values by setting **cmd** to 0x0006. After it is started, the child process returns control to the parent process if it encounters a break point, a non-maskable interrupt, a single-step interrupt, or the end of the program.

The **DosPTrace** function can be used to debug a process with multiple threads by setting the **tid** field of the **PTRACEBUF** structure to the identifier of the thread to be debugged. Other threads in the process are suspended. (The address space is the same for all threads in a process.) Commands to read from or write to memory locations or set break points affect all threads in the process, even if the command is issued with a specific thread identifier. If the parent process uses the 0x000B command, a selected thread or group of threads can keep running while others are suspended. This allows only the selected threads to be affected by the break points and manipulated.

See Also

DosExecPgm, DosGetInfoSeg

DosPurgeQueue

USHORT DosPurgeQueue(*hqueue*)
HQUEUE *hqueue*; /* handle of queue to be purged */

The **DosPurgeQueue** function purges a queue of all elements.

Parameters

hqueue Identifies the queue to be purged. This handle must have been created previously by using the **DosCreateQueue** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_QUE_INVALID_HANDLE
ERROR_QUE_PROC_NOT_OWNED

Comments

Only the process that created the queue may call the **DosPurgeQueue** function.

See Also

DosCreateQueue

DosPutMessage

```
USHORT DosPutMessage(hf, cbMsg, pchMsg)
HFILE hf;          /* handle of output file/device */
USHORT cbMsg;      /* length of message buffer     */
PCHAR pchMsg;      /* pointer to message buffer    */
```

The **DosPutMessage** function writes the message pointed to by the *pchMsg* parameter to the file identified by the *hf* parameter.

The **DosPutMessage** function is a family API function.

Parameters

hf Identifies the file that receives the message. This handle must have been created previously by using the **DosOpen** function. Standard file handles (such as 1 and 2) can also be used.

cbMsg Specifies the length (in bytes) of the message to output.

pchMsg Points to the message to output.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_HANDLE
ERROR_MR_UN_PERFORM
ERROR_WRITE_PROTECT

Comments

The **DosPutMessage** function uses an 80-column screen width. If a word is about to span column 80, the function "wraps" the word to a new line at column 1. If the last character to be positioned on a line is a double-byte character that would be bisected, this rule ensures that the character is not bisected.

When handling word wrapping, the **DosPutMessage** function uses column 1 as the starting position of the cursor.

Restrictions

In real mode, the following restriction applies to the **DosPutMessage** function:

- There is no method of identifying the boot drive. The system assumes that the message file is in the root directory of the current drive.

See Also

DosGetMessage, DosInsMessage, DosOpen

DosQAppType

```
USHORT DosQAppType(pszPrgName, pusType)
PSZ pszPrgName;       /* pointer to executable-file name  */
PUSHORT pusType;      /* pointer to application-type flags */
```

The **DosQAppType** function retrieves the application type of an executable file. The application type is specified at link time in the module-definition file.

Parameters

pszPrgName Points to the null-terminated string that contains the name of the executable file for which the flags are to be returned. If the string appears to be a fully qualified path (that is, it contains a colon in the second position and/or contains a backslash), the file will be searched for in the indicated directory on the indicated drive. If neither of these conditions is true and the file is not in the current directory, each drive and directory specification in the path defined in the current program's environment will be searched. The default extension for an executable file is *.exe*, although any extension is acceptable.

pusType Points to a word containing flags that specify the application type, as determined from the header of the executable file specified by the *pszPrgName* parameter. Upon return, the variable pointed to by the *pusType* parameter will have one or more of the following flags set:

Value	Meaning
BOUND	Application has been "bound" and can run either in protected mode or with MS-DOS (either the compatibility box or MS-DOS, version 3.*x*).
DOSFORMAT	Application will only run with MS-DOS.
DYNAMICLINK	Application is a dynamic-link module.
NOTSPECIFIED	Application type is not specified in executable header.
NOTWINDOCOMPAT	Application will run only in a full screen session.
WINDOWAPI	Application runs as a Presentation Manager window.
WINDOWCOMPAT	Application will run in a VIO window.

Return Value

The return value is zero if the function is successful. Otherwise, it is one of the following values:

ERROR_BAD_FORMAT
ERROR_DRIVE_LOCKED
ERROR_EXE_MARKED_INVALID
ERROR_FILE_NOT_FOUND
ERROR_INVALID_EXE_SIGNATURE
ERROR_TOO_MANY_OPEN_FILES

■ DosQCurDir

```
USHORT DosQCurDir(usDriveNumber, pszPathBuf, pcbPathBuf)
USHORT usDriveNumber;   /* drive number                              */
PBYTE pszPathBuf;       /* pointer to buffer receiving directory path   */
PUSHORT pcbPathBuf;     /* pointer to variable receiving length of path */
```

The **DosQCurDir** function retrieves the path of the current directory on the specified drive. **DosQCurDir** copies a null-terminated string identifying the current directory to the buffer pointed to by the *pszPathBuf* parameter. The string consists of one or more directory names separated by backslashes (\). The drive letter is not part of the returned string.

The **DosQCurDir** function is a family API function.

Parameters

usDriveNumber Specifies the drive number. The default drive is 0, drive A is 1, drive B is 2, and so on.

pszPathBuf Points to a buffer that receives the path of the current directory. The path of the current directory is copied to this buffer only if the buffer is large enough to contain the complete directory.

pcbPathBuf Points to the variable that contains the size (in bytes) of the *pszPathBuf* buffer. If the buffer is too small to contain the current path, the error value ERROR_BUFFER_OVERFLOW is returned and this variable receives the size of the buffer required to contain the complete pathname.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BUFFER_OVERFLOW
ERROR_DRIVE_LOCKED
ERROR_INVALID_DRIVE
ERROR_NOT_DOS_DISK
ERROR_NOT_READY

Example

This example calls the **DosQCurDisk** function to retrieve the current drive number, sets the buffer length to zero, and calls **DosQCurDir**. Since the buffer is too small to contain a path of any size, **DosQCurDir** returns the size needed in the *cbPath* variable. The **DosAllocSeg** function is called to allocate the memory needed for the buffer, and **DosQCurDir** is called again to retrieve the path name. This method of setting the buffer length will be successful in any version of MS OS/2, including future versions, in which the maximum path length may be longer.

```
PSZ pszPath;
USHORT cbPath, usDisk;
ULONG ulDrives;
SEL selPath;

cbPath = 0;
DosQCurDisk(&usDisk, &ulDrives);      /* gets current drive          */

/* First call DosQCurDir to find out the size of the buffer needed. */

DosQCurDir(usDisk, NULL, &cbPath);
DosAllocSeg(cbPath, &selPath, SEG_NONSHARED);   /* allocates memory */
pszPath = MAKEP(selPath, 0);          /* assigns it to a far pointer */
DosQCurDir(usDisk,                    /* drive number                */
    pszPath,                          /* buffer for directory path   */
    &cbPath);                         /* length of directory buffer  */
```

See Also

DosChDir, DosQCurDisk, DosSelectDisk

■ DosQCurDisk

USHORT DosQCurDisk(*pusDriveNumber, pulLogicalDrives***)**
PUSHORT *pusDriveNumber*; /* pointer to variable receiving drive number */
PULONG *pulLogicalDrives*; /* pointer to variable receiving drive map */

The **DosQCurDisk** function retrieves the current drive number and a map of the logical drives.

The **DosQCurDisk** function is a family API function.

Parameters

pusDriveNumber Points to the variable that receives the number of the default drive. For example, drive A is 1, drive B is 2, and so on.

pulLogicalDrives Points to the variable that receives the map of the logical drive.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The current drive number identifies the disk drive to be searched for a given file if no explicit drive name is given when the filename is specified. The current drive number is used by functions such as **DosOpen** and **DosFindFirst**. Each process has its own current drive and may change this drive, by using the **DosChDir** function, without affecting other processes. The default current drive for a process is the drive on which the process is called.

The map of the logical drives identifies which of the 26 possible disk drives exist. The map is a 32-bit value in which each bit of the low-order 26 bits represents a single drive. For example, bit 0 represents drive A, bit 1 represents drive B, and so on. If a bit is set to 1, the drive exists; if it is cleared to 0, the drive does not exist.

Example

This example calls the **DosQCurDisk** function to determine the current default drive and how many logical drives exist. The example then displays the letter of every logical drive after checking whether its bit is set in the *ulDrives* variable.

```
CHAR chDrives;
USHORT usDisk;
ULONG ulDrives;
DosQCurDisk(&usDisk, &ulDrives);           /* gets current drive       */
for (chDrives = 'A'; chDrives <= 'Z'; chDrives++) {
    if (ulDrives & 1)                      /* if the drive bit is set, */
        VioWrtTTY(&chDrives, 1, 0);        /* displays the drive letter */
    ulDrives >>= 1;
}
```

See Also

DosChDir, DosFindFirst, DosOpen, DosQCurDir, DosSelectDisk

■ DosQFHandState

```
USHORT DosQFHandState(hf, pfsOpenMode)
HFILE hf;                  /* file handle                          */
PUSHORT pfsOpenMode;       /* pointer to variable for file-handle state */
```

The **DosQFHandState** function retrieves the state of the specified file handle. The file-handle state indicates whether the file may be read from or written to and whether it may be opened for reading or writing by other processes.

The **DosQFHandState** function is a family API function.

Parameters

hf Identifies the file whose file-handle state is to be retrieved. This handle must have been previously created by using the **DosOpen** function.

pfsOpenMode Points to the variable that receives the file-handle state. The file-handle state consists of one access mode, one share mode, and optional flags. It is identical to the values specified in the *fsOpenMode* parameter of the **DosOpen** function. Which values are set can be determined by using the AND operator to combine the value returned in the *pfsOpenMode* parameter with one or more of the following values:

Value	Meaning
OPEN_ACCESS_READONLY	Data may be read from the file but not written to it.
OPEN_ACCESS_READWRITE	Data may be read from or written to the file.
OPEN_ACCESS_WRITEONLY	Data may be written to the file but not read from it.
OPEN_SHARE_DENYNONE	Other processes can open the file for any access: read-only, write-only, or read-write.
OPEN_SHARE_DENYREAD	Other processes can open the file for write-only access but they cannot open it for read-only or read-write access.
OPEN_SHARE_DENYREADWRITE	The current process has exclusive access to the file.
OPEN_SHARE_DENYWRITE	Other processes can open the file for read-only access but they cannot open it for write-only or read-write access.
OPEN_FLAGS_DASD	The file handle represents a physical drive that has been opened for direct access.
OPEN_FLAGS_FAIL_ON_ERROR	Any function that uses the file handle returns immediately with an error code if there is an I/O error.
OPEN_FLAGS_NOINHERIT	The file handle is private to the current process.
OPEN_FLAGS_WRITE_THROUGH	The system writes data to the device before the given function returns.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_HANDLE

Example

This example calls the **DosQFHandState** function using the handle of a previously opened file, and then checks the *fsOpenMode* variable and reports if the file is opened for read/write access:

```
HFILE hf;
USHORT fsOpenMode;
    .
    .
    .
DosQFHandState(hf, &fsOpenMode);
if (fsOpenMode & OPEN_ACCESS_READWRITE)
    VioWrtTTY("File opened for read/write access\r\n", 35, 0);
if (fsOpenMode & OPEN_SHARE_DENYREADWRITE)
    VioWrtTTY("File cannot be shared\r\n", 23, 0);
```

See Also

DosDevIOCtl, DosExecPgm, DosOpen, DosSetFHandState

DosQFileInfo

```
USHORT DosQFileInfo(hf, usInfoLevel, pfstsInfo, cbInfoBuf)
HFILE hf;                /* handle of file about which data sought */
USHORT usInfoLevel;      /* level of file data required            */
PFILESTATUS pfstsInfo;   /* pointer to file-data buffer            */
USHORT cbInfoBuf;        /* length of file-data buffer             */
```

The **DosQFileInfo** function retrieves information about a specific file. The file information consists of the date and time the file was created, the date and time it was last accessed, the date and time it was last written to, the size of the file, and its attributes.

The file information is based on the most recent call to the **DosClose** or **DosBufReset** function.

The **DosQFileInfo** function is a family API function.

Parameters

hf Identifies the file about which information is to be retrieved. This handle must have been created previously by using the **DosOpen** function.

usInfoLevel Specifies the level of file information required. In MS OS/2, version 1.1, this value must be 0x0001.

pfstsInfo Points to the structure that receives the file information. The **FILESTATUS** structure has the following form:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

cbInfoBuf Specifies the length (in bytes) of the buffer that receives the file information.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BUFFER_OVERFLOW
ERROR_DIRECT_ACCESS_HANDLE
ERROR_INVALID_HANDLE
ERROR_INVALID_LEVEL

Example

This example opens the file *abc*, calls the **DosQFileInfo** function to retrieve the current allocated size, and then calls the **DosNewSize** function to increase the file's size by 1K:

```
HFILE hf;
USHORT usAction;
FILESTATUS fstsFile;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, 0L);
DosQFileInfo(hf,                 /* file handle                 */
    1,                           /* level of information        */
    (PBYTE) &fstsFile,           /* address of file-data buffer */
    sizeof(fstsFile));           /* size of data buffer         */
DosNewSize(hf, fstsFile.cbFileAlloc + 1024L);
```

See Also

DosBufReset, DosClose, DosOpen, DosQFileMode, DosSetFileInfo

■ DosQFileMode

```
USHORT DosQFileMode(pszFileName, pusAttribute, ulReserved)
PSZ pszFileName;          /* pointer to filename                  */
PUSHORT pusAttribute;     /* pointer to variable for file attributes */
ULONG ulReserved;         /* must be zero                         */
```

The **DosQFileMode** function retrieves the attributes (mode) of the specified file. The file attributes are set when the file is created and can be changed at any time by using the **DosSetFileMode** function.

The **DosQFileMode** function is a family API function.

Parameters

pszFileName Points to a null-terminated string that specifies the name of the file to be checked. The string must be a valid MS OS/2 filename.

pusAttribute Points to the variable that receives the file attributes. It can be one or more of the following values:

Value	Meaning
FILE_NORMAL	File can be read from and written to.
FILE_READONLY	File can be read from but not written to.
FILE_HIDDEN	File is hidden and does not appear in a directory listing.
FILE_SYSTEM	File is a system file.
FILE_DIRECTORY	File is a subdirectory.
FILE_ARCHIVED	File has been archived.

ulReserved Specifies a reserved value; must be zero.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND

Comments You cannot use the **DosQFileMode** function to retrieve the attributes of the volume label. The attributes of a volume label can be retrieved by using the **DosQFSInfo** function.

Example This example calls the **DosQFileMode** function and displays a message if the filename *abc* is a subdirectory:

```
USHORT usAttribute;
DosQFileMode("abc",        /* filename                          */
    &usAttribute,          /* address of variable for file's attribute */
    0L);                   /* reserved                          */
if (usAttribute == FILE_DIRECTORY)
    VioWrtTTY("abc is a subdirectory\r\n", 23, 0);
```

See Also **DosQFHandState, DosQFSInfo, DosSetFileMode**

DosQFSInfo

```
USHORT DosQFSInfo(usDriveNumber, usInfoLevel, pbInfo, cbInfo)
USHORT usDriveNumber;    /* drive number                     */
USHORT usInfoLevel;      /* type of information              */
PBYTE pbInfo;            /* pointer to buffer for information */
USHORT cbInfo;           /* length of information buffer     */
```

The **DosQFSInfo** function retrieves file-system information from the disk in the specified drive. This file-system information defines characteristics of the disk, such as its size.

There are two levels of file-system information. Level 1 file-system information specifies the number of sectors per allocation unit on the disk, the number of allocation units, the available allocation units, and the number of bytes per sector. Level 2 file-system information defines the volume label and the date and time at which the label was created.

The **DosQFSInfo** function is a family API function.

Parameters *usDriveNumber* Specifies the logical drive number for the disk about which information is to be retrieved. This parameter can be any value from 0 through 26. If this parameter is zero, information about the disk in the current drive is retrieved. Otherwise, 1 specifies drive A, 2 specifies drive B, and so on.

usInfoLevel Specifies the level of file information to be retrieved. In MS OS/2, version 1.1, this value can be 1 or 2.

pbInfo Points to the structure that receives the file-system information. For level 1 information, it points to an **FSALLOCATE** structure. For level 2, it points to an **FSINFO** structure. An **FSALLOCATE** structure has the following form:

```
typedef struct _FSALLOCATE {
    ULONG  idFileSystem;
    ULONG  cSectorUnit;
    ULONG  cUnit;
    ULONG  cUnitAvail;
    USHORT cbSector;
} FSALLOCATE;
```

An **FSINFO** structure has the following form:

```
typedef struct _FSINFO {
    FDATE fdateCreation;
    FTIME ftimeCreation;
    VOLUMELABEL vol;
} FSINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

cbInfo Specifies the length (in bytes) of the buffer that receives the file-system information.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BUFFER_OVERFLOW
ERROR_INVALID_DRIVE
ERROR_INVALID_LEVEL
ERROR_NO_VOLUME_LABEL

Example

This example calls the **DosQFSInfo** function and displays the volume label of drive C:

```
FSINFO fsinf;
DosQFSInfo(3,                          /* drive number (c:)            */
    2,                                 /* level of information requested */
    (PBYTE) &fsinf,                    /* address of buffer            */
    sizeof(FSInfoBuf));                /* size of buffer               */
VioWrtTTY(fsinf.vol.szVolLabel, fsinf.vol.cch, 0);
```

See Also

DosQFHandState, DosQFileMode, DosSetFSInfo

DosQHandType

```
USHORT DosQHandType(hf, pfsType, pusDeviceAttr)
HFILE hf;                 /* file handle                                */
PUSHORT pfsType;          /* pointer to variable for handle type        */
PUSHORT pusDeviceAttr;    /* pointer to variable for device attribute   */
```

The **DosQHandType** function retrieves information that specifies whether the given file handle identifies a file, device, or pipe.

Parameters

hf Identifies the file. This handle must have been created previously by using the **DosOpen** function.

pfsType Specifies the type of file or device associated with the file handle. It can be one of the following:

Value	Meaning
HANDTYPE_DEVICE	The handle is to a device, such as a printer.
HANDTYPE_FILE	The handle is to a file.
HANDTYPE_PIPE	The handle is to a pipe.

If the file or device is located on a network, this parameter is a combination of one of the values given above and the value HANDTYPE_NETWORK (0x8000).

pusDeviceAttr Points to the variable that receives the device-driver attribute word.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_HANDLE

Comments

The **DosQHandType** function allows some interactive or file-oriented programs to determine the source of their input. For example, the *cmd.exe* program suppresses the system prompt if the input is from a disk file.

Example

This example calls the **DosQHandType** function to determine if standard output has been redirected to a file. The LOBYTE macro is an important part of this example; it allows the handle type to be determined even if the handle is to a file or device on a network:

```
USHORT fsType, usDeviceAttr;
DosQHandType(1,                                  /* file handle      */
    &usHandType,                                 /* type of handle   */
    &usDeviceAttr);                              /* device attribute */
if (LOBYTE(fsType) == HANDTYPE_DEVICE)
    VioWrtTTY("stdout is a device\r\n", 20, 0);
else if (LOBYTE(fsType) & HANDTYPE_FILE) {
    if (fsType & HANDTYPE_NETWORK)
        VioWrtTTY("stdout is a networked file\r\n", 28, 0);
    else
        VioWrtTTY("stdout is a local file\r\n", 24, 0);
}
```

See Also

DosOpen, DosQFHandState

DosQNmPHandState

USHORT DosQNmPHandState (*hp, pfsState*)

```
HPIPE hp;              /* pipe handle                                */
PUSHORT pfsState;      /* pointer to variable receiving handle state */
```

The **DosQNmPHandState** function retrieves information about the state of a specified pipe handle.

Parameters

hp Identifies the pipe to read from.

pfsState Points to the variable that receives the handle state. This parameter is a combination of an instance count, a read-mode flag, a type flag, an end-point flag, and a wait flag. The possible values are:

Value	Meaning
PIPE_END_CLIENT	The handle is the client end of a named pipe.
PIPE_END_SERVER	The handle is the server end of a named pipe.
PIPE_NOWAIT	Reading from the pipe returns immediately if no data is available. If this flag is not set, reading from the pipe waits until data is available.
PIPE_READMODE_MESSAGE	Read the pipe as a message stream. If this flag is not set, the pipe is read as a byte stream.
PIPE_TYPE_MESSAGE	The pipe is a message-stream pipe. If this flag is not set, the pipe is a byte-stream pipe.
PIPE_UNLIMITED_INSTANCES	Unlimited instances of the pipe can be created. If this flag is not specified, a value from 1 through 254 can be used for the number of instances.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_PIPE_NOT_CONNECTED

Comments

If the handle is the server end of the pipe, the handle-state values are identical to those set by the call to the **DosMakeNmPipe** function that created the pipe. If the handle is the client end of the pipe, the handle-state values are determined by the **DosOpen** function that opened the pipe or are set by the **DosSetNmPHandState** function.

See Also

DosMakeNmPipe, DosOpen, DosSetNmPHandState

DosQNmPipeInfo

```
USHORT DosQNmPipeInfo(hp, usInfoLevel, pbBuf, cbBuf)
HPIPE hp;                /* pipe handle                          */
USHORT usInfoLevel;      /* level of information to retrieve     */
PBYTE pbBuf;             /* pointer to buffer receiving information */
USHORT cbBuf;            /* number of bytes in buffer            */
```

The **DosQNmPipeInfo** function retrieves information about a named pipe.

Parameters

hp Identifies the pipe to read from.

usInfoLevel Specifies the level of information to retrieve. Level 1 is miscellaneous information about the pipe. Level 2 identifies the pipe's clients.

pbBuf Points to the buffer that receives the information. For level–2 information, the buffer will contain a unique 2-byte identifier of the client. For level–1 information, the data is stored in the **PIPEINFO** structure, which has the following form:

```
typedef struct _PIPEINFO {
    USHORT cbOut;
    USHORT cbIn;
    BYTE   cbMaxInst;
    BYTE   cbCurInst;
    BYTE   cbName;
    CHAR   szName[1];
} PIPEINFO;
```

For more information, see Chapter 4, "Types, Macros, Structures."

cbBuf Specifies the size (in bytes) of the buffer receiving the information.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_BUFFER_OVERFLOW
ERROR_INVALID_LEVEL
ERROR_INVALID_PARAMETER
ERROR_PIPE_NOT_CONNECTED

See Also

DosQNmPHandState, DosQNmPipeSemState

DosQNmPipeSemState

```
USHORT DosQNmPipeSemState(hsem, pbBuf, cbBuf)
HSEM hsem;      /* semaphore handle                   */
PBYTE pbBuf;    /* pointer to buffer receiving information */
USHORT cbBuf;   /* buffer size                        */
```

The **DosQNmPipeSemState** function returns information about all local named pipes that are in blocking mode and are associated with a specified system semaphore.

Parameters

hsem Identifies the semaphore that is associated with the named pipe.

pbBuf Points to the buffer that receives the information.

cbBuf Specifies the length (in bytes) of the buffer that receives the information.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_PARAMETER
ERROR_SEM_NOT_FOUND

See Also

DosSetNmPipeSem

DosQSysInfo

```
USHORT DosQSysInfo(index, pbSysInfoBuf, cbSysInfoBuf)
USHORT index;              /* index of value to look up                      */
PBYTE pbSysInfoBuf;        /* pointer to buffer receiving information        */
USHORT cbSysInfoBuf;       /* number of bytes in buffer receiving information */
```

The **DosQSysInfo** function retrieves system-format information, such as maximum path length, that is constant for a particular release of MS OS/2.

Parameters

index Specifies the index of the information to retrieve. In MS OS/2, version 1.1, the only available index is zero, which returns the maximum path length (including the trailing null character).

pbSysInfoBuf Points to the buffer that receives the system information. When the value of the *index* is zero, the **DosQSysInfo** function puts the maximum path length into the first two bytes of the buffer.

cbSysInfoBuf Specifies the length (in bytes) of the buffer to receive the system information.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BUFFER_OVERFLOW
ERROR_INVALID_PARAMETER

See Also

DosGetVersion

DosQueryQueue

```
USHORT DosQueryQueue(hqueue, pusElemCount)
HQUEUE hqueue;             /* queue handle                    */
PUSHORT pusElemCount;      /* pointer to variable for element count */
```

The **DosQueryQueue** function retrieves a count of the number of elements in the given queue. Any process that has a queue open can call this function.

Parameters

hqueue Identifies the queue about which information is sought. This handle must have been previously created or opened by using the **DosCreateQueue** or **DosOpenQueue** function.

pusElemCount Points to the variable that receives the count of elements in the queue.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_QUE_INVALID_HANDLE

See Also

DosCreateQueue, **DosOpenQueue**

DosQVerify

USHORT DosQVerify(*pfVerifyOn* **)**
PBOOL *pfVerifyOn*; /* verification-mode indicator */

The **DosQVerify** function retrieves the verification mode. The verification mode specifies whether the system verifies the data each time it writes data to a disk.

The **DosQVerify** function is a family API function.

Parameters

pfVerifyOn Points to the variable that receives the verification mode. The *pfVerifyOn* parameter is set to TRUE if the system verifies the data. Otherwise, it is set to FALSE.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Example

This example calls the **DosQVerify** function to determine if write verification is active and then displays the result:

```
BOOL fVerifyOn;
DosQVerify(&fVerifyOn);
if (fVerifyOn == TRUE)
    VioWrtTTY("Verify mode is active\r\n", 23, 0);
else
    VioWrtTTY("Verify mode is not active\r\n", 27, 0);
```

See Also

DosSetVerify

DosR2StackRealloc

USHORT DosR2StackRealloc(*usSize* **)**
USHORT *usSize*; /* new size for stack */

The **DosR2StackRealloc** function changes the size of a thread's ring–2 stack. The function reallocates the stack as requested.

This function cannot be used from ring 2.

Parameters

usSize Specifies the size (in bytes) of the ring–2 stack. The new stack size cannot be less than the current stack size.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

DosRead

```
USHORT DosRead(hf, pvBuf, cbBuf, pcbBytesRead)
HFILE hf;                    /* file handle                          */
PVOID pvBuf;                 /* pointer to buffer receiving data     */
USHORT cbBuf;                /* number of bytes in buffer            */
PUSHORT pcbBytesRead;        /* pointer to variable for number of bytes read */
```

The **DosRead** function reads up to a specified number of bytes of data from a file into a buffer. The function may read fewer than the specified number of bytes if it reaches the end of the file.

The **DosRead** function is a family API function.

Parameters

hf Identifies the file to be read. This handle must have been created previously by using the **DosOpen** function.

pvBuf Points to the buffer that receives the data.

cbBuf Specifies the number of bytes to read from the file.

pcbBytesRead Points to the variable that receives the number of bytes read from the file. This parameter is zero if the file pointer is positioned at the end of the file prior to the call to the **DosRead** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NOT_DOS_DISK

Comments

The **DosRead** function does not return an error if the file pointer is at the end of the file when the read operation begins.

Example

This example opens, reads, and displays the file *abc*:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesRead, cbBytesWritten;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
do {
    DosRead(hf,             /* file handle                    */
        abBuf,              /* address of buffer              */
        sizeof(abBuf),      /* size of buffer                 */
        &cbBytesRead);      /* address for number of bytes read */
    DosWrite(1, abBuf, cbBytesRead, &cbBytesWritten);
}
while (cbBytesRead);
```

See Also

DosChgFilePtr, DosOpen, DosReadAsync, DosWrite, KbdStringIn

DosReadAsync

```
USHORT DosReadAsync(hf, hsemRam, pusErrCode, pvBuf, cbBuf, pcbBytesRead)
HFILE hf;               /* file handle                                   */
PULONG hsemRam;         /* pointer to RAM semaphore                      */
PUSHORT pusErrCode;     /* pointer to variable for error return code     */
PVOID pvBuf;            /* pointer to input buffer                       */
USHORT cbBuf;           /* length of input buffer                        */
PUSHORT pcbBytesRead;   /* pointer to variable for number of bytes read */
```

The **DosReadAsync** function reads one or more bytes of data from the file identified by the *hf* parameter. The function reads the data asynchronously; that is, the function returns immediately to the process that called it but continues to copy data to the specified buffer while the execution of the process continues.

Parameters

hf Identifies the file to be read. This handle must have been previously opened by using the **DosOpen** function.

hsemRam Points to the RAM semaphore that indicates when the function has finished reading the data.

pusErrCode Points to the variable that receives any error code the function generates while reading data. The possible error codes are identical to those returned by the **DosRead** function.

pvBuf Points to the buffer that receives the data being read.

cbBuf Specifies the number of bytes to be read from the file identified by the *hf* parameter.

pcbBytesRead Points to the variable that receives the number of bytes read from the file.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NO_PROC_SLOTS
ERROR_NOT_DOS_DISK

Comments

The **DosReadAsync** function reads up to the number of bytes specified in the *cbBuf* parameter, but it may read fewer if it reaches the end of the file. In any case, the function copies the number of bytes read to the variable pointed to by the *pcbBytesRead* parameter. The *pcbBytesRead* parameter is zero if all the bytes in the file have been read (that is, the end of file has been reached).

If the process intends to use the RAM semaphore pointed to by the *hsemRam* parameter to determine when data is available, it must set the semaphore by using the **DosSemSet** function before calling **DosReadAsync**. When **DosReadAsync** has read the data, it clears the RAM semaphore.

The **DosReadAsync** function carries out the asynchronous operation by creating a new thread that reads from the specified file. The function terminates the thread when the operation is complete or when an error occurs.

Example

This example opens the file *abc*, sets a RAM semaphore, and calls the **DosReadAsync** function to read part of the file. While the file is being read, program execution continues until the call to the **DosSemWait** function, which does not return until the **DosReadAsync** thread completes its work.

```
BYTE abBuf[512];
ULONG hReadSemaphore = 0;
HFILE hf;
USHORT usAction, cbBytesRead;
USHORT usReadReturn;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
DosSemSet(&hReadSemaphore);      /* sets RAM semaphore          */
DosReadAsync(hf,                 /* handle to file              */
    &hReadSemaphore,             /* address of semaphore        */
    &usReadReturn,               /* address to store return code */
    abBuf,                       /* address of buffer           */
    sizeof(abBuf),               /* size of buffer              */
    &cbBytesRead);               /* number of bytes read        */
    .
    . /* other processing takes place here */
    .
DosSemWait(&hReadSemaphore, -1L);
```

See Also

DosOpen, DosRead, DosSemSet, DosSemWait, DosWriteAsync

DosReadQueue

USHORT DosReadQueue(*hqueue, pqresc, pcbElement, ppv, usElement, fWait, pbElemPrty, hsem***)**

```
HQUEUE hqueue;              /* handle of queue to read                     */
PQUEUERESULT pqresc;        /* pointer to structure for PID and request code */
PUSHORT pcbElement;         /* pointer to variable for length of element   */
PVOID FAR * ppv;            /* pointer to buffer for element               */
USHORT usElement;           /* element number to read                      */
UCHAR fWait;                /* wait/no wait indicator                      */
PBYTE pbElemPrty;           /* pointer to variable for priority of element */
HSEM hsem;                  /* semaphore handle                            */
```

The **DosReadQueue** function retrieves an element from a queue and removes it from the queue. It copies the element to the buffer pointed to by the *ppv* parameter and fills the structure pointed to by the *pqresc* parameter with information about the element.

Parameters

hqueue Identifies the queue to be read. This handle must have been previously created or opened by using the **DosCreateQueue** or **DosOpenQueue** function.

pqresc Points to the structure that receives information about the request. The **QUEUERESULT** structure has the following form:

```
typedef struct _QUEUERESULT {
    PID pidProcess;
    USHORT usEventCode;
} QUEUERESULT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pcbElement Points to the variable that receives the length in bytes of the element.

ppv Points to a pointer that receives the address of the element in the queue.

usElement Specifies where to look in the queue for the element. If the *usElement* parameter is 0x0000, the function looks at the beginning of the queue. Otherwise, the function assumes the value is an element identifier retrieved by using the **DosPeekQueue** function and looks for the specified element.

fWait Specifies whether to wait for an element to be placed in the queue, if the queue is empty. If the *fWait* parameter is DCWW_WAIT, the function waits until an element is available. If it is DCWW_NOWAIT, the function returns immediately with a code that indicates there are no entries in the queue.

pbElemPrty Points to a variable that receives the priority value specified when the element was added to the queue. This is a numeric value from 0 through 15; 15 is the highest priority.

hsem Identifies a semaphore. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore. This semaphore would typically be used in a call to the **DosMuxSemWait** function to wait until the queue has an element. If the *fWait* parameter is DCWW_WAIT, *hsem* is ignored.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_QUE_ELEMENT_NOT_EXIST
ERROR_QUE_EMPTY
ERROR_QUE_INVALID_HANDLE
ERROR_QUE_INVALID_WAIT
ERROR_QUE_PROC_NOT_OWNED

Comments

If the queue is empty, the **DosReadQueue** function either returns immediately or waits for an element to be written to the queue, depending on the value of the *fWait* parameter.

Only the process that created the queue may call the **DosReadQueue** function.

See Also

DosCreateQueue, DosMuxSemWait, DosOpenQueue, DosPeekQueue, DosWriteQueue

■ DosReallocHuge

```
USHORT DosReallocHuge(usNumSeg, usPartialSeg, sel)
USHORT usNumSeg;        /* number of 65, 536-byte segments */
USHORT usPartialSeg;    /* number of bytes in last segment */
SEL sel;                /* segment selector                */
```

The **DosReallocHuge** function reallocates a huge memory block. The function changes the size of the huge memory to the number of 65,536-byte segments specified by the *usNumSeg* parameter plus an additional segment of the size specified by the *usPartialSeg* parameter.

The **DosReallocHuge** function is a family API function.

Parameters

usNumSeg Specifies the number of 65,536-byte segments to allocate.

usPartialSeg Specifies the number of bytes in the last segment. This number can be any value from 0 through 65,535. If it is zero, no additional segment is allocated.

sel Specifies the selector for the huge memory block to be reallocated. The selector must have been created previously by using the **DosAllocHuge** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosReallocHuge** function does not change the sharable and discardable attributes of the segments in the huge memory block. If it was originally a sharable or discardable block, it remains a sharable or discardable block. However, if **DosReallocHuge** reallocates a discardable block, it also locks the segments. The **DosUnlockSeg** function must be used to unlock the segments and permit discarding.

The memory block cannot be reallocated for a size larger than the maximum specified by the *usMaxNumSeg* parameter in the original call to the **DosAllocHuge** function.

Each segment in the huge memory block has a unique selector. The selectors are consecutive. The *sel* parameter specifies the value of the first selector; the remaining selectors can be computed by adding the selector offset to the first selector one or more times—that is, once for the second selector, twice for the third, and so on. The selector offset is a multiple of 2, as specified by the shift count retrieved by using the **DosGetHugeShift** function. For example, if the shift count is 2, the selector offset is 4 (1 << 2). If the selector offset is 4 and the first selector is 6, the second selector is 10, the third is 14, and so on.

Restrictions

In real mode, the following restriction applies to the **DosReallocHuge** function:

- The *usPartialSeg* parameter is rounded up to the next paragraph (16-byte) value.

See Also

DosAllocHuge, DosFreeSeg, DosGetHugeShift, DosLockSeg, DosReallocSeg, DosUnlockSeg

DosReallocSeg

```
USHORT DosReallocSeg(usNewSize, sel)
USHORT usNewSize;     /* new segment size */
SEL sel;              /* segment selector */
```

The **DosReallocSeg** function reallocates a segment. The function changes the size of the segment to the number of bytes specified in the *usNewSize* parameter.

The **DosReallocSeg** function is a family API function.

Parameters

usNewSize Specifies the new size (in bytes). The size can be any number from 0 through 65,535. If it is zero, the function allocates 65,536 bytes.

sel Specifies the selector of the segment to be reallocated. The selector must have been created previously by using the **DosAllocSeg** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosReallocSeg** function does not change the sharable and discardable attributes of the segment. If it was originally a sharable or discardable segment, it remains a sharable or discardable segment.

However, if **DosReallocSeg** reallocates a discardable segment, it also locks the segment. You must use the **DosUnlockSeg** function to unlock the segment and permit discarding.

The **DosReallocSeg** function cannot reallocate a shared segment to a size smaller than its original size.

Restrictions

In real mode, the following restriction applies to the **DosReallocSeg** function:

- The *usNewSize* parameter is rounded up to the next paragraph (16-byte) value.

See Also

DosAllocSeg, DosFreeSeg, DosLockSeg, DosReallocHuge, DosUnlockSeg

■ DosResumeThread

```
USHORT DosResumeThread(tid)
TID tid;      /* identifier of thread to be resumed */
```

The **DosResumeThread** function restarts a thread that was previously stopped by the **DosSuspendThread** function.

Parameters

tid Specifies the thread identifier of the thread to be resumed. The thread must have been created previously by using the **DosCreateThread** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_THREADID

See Also

DosCreateThread, DosSuspendThread

■ DosRmDir

USHORT DosRmDir(*pszDirName, ulReserved***)**
PSZ *pszDirName*; /* directory name */
ULONG *ulReserved*; /* must be zero */

The **DosRmDir** function removes the specified directory. The directory must be empty before it can be removed; that is, it must not contain files of any kind, including hidden files and other directories. If the specified directory cannot be found or is not empty, **DosRmDir** returns an error.

The **DosRmDir** function is a family API function.

Parameters

pszDirName Points to a null-terminated string that specifies the directory to be removed. This string must be a valid MS OS/2 directory name.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_CURRENT_DIRECTORY
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND

Comments

The **DosRmDir** function cannot remove the current directory or the root directory.

If necessary, use the **DosDelete** function to remove files from the directory.

Example

This example deletes all files in the subdirectory *abc* and then calls the **DosRmDir** function to delete the subdirectory. If the subdirectory contains other subdirectories or files that cannot be deleted, the **DosRmDir** function returns an error.

```
USHORT usError;

DosDelete("abc\\*.*", 0L);          /* removes all files    */
usError = DosRmDir("abc", 0L);      /* removes subdirectory */
if (usError)
    VioWrtTTY("Can't delete subdirectory\r\n", 27, 0);
else {
```

See Also

DosChDir, DosDelete, DosMkDir

DosScanEnv

```
USHORT DosScanEnv(pszVarName, ppszResult)
PSZ pszVarName;          /* pointer to environment-variable name */
PSZ FAR * ppszResult;    /* pointer to variable for result pointer  */
```

The **DosScanEnv** function searches an environment for a specified environment variable. The environment is one or more null-terminated strings that name and define the environment variables available to the current process. Environment variables can be used to pass information to a program—for example, a variable might name a list of directories that contain data files to be used by the program.

An environment variable has the following form:

name=value

The **DosScanEnv** function searches for the environment variable whose name matches the name pointed to by the *pszVarName* parameter. If **DosScanEnv** finds the variable, it copies the address of the first character of the environment variable's value to the variable pointed to by the *ppszResult* parameter. The first character of the environment variable's value is the character following the equal sign (=).

Parameters

pszVarName Points to a null-terminated string that specifies the name of an environment variable. The string must not include a trailing equal sign (=), since the equal sign is not part of the name.

ppszResult Points to the pointer variable that receives the address of the environment string.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

DosExecPgm, DosGetEnv, DosSearchPath

DosSearchPath

```
USHORT DosSearchPath(fsSearch, pszPath, pszFileName, pbBuf, cbBuf)
USHORT fsSearch;      /* search flags                                */
PSZ pszPath;          /* pointer to search path or environment variable */
PSZ pszFileName;      /* pointer to filename                         */
PBYTE pbBuf;          /* pointer to result buffer                    */
USHORT cbBuf;         /* length of result buffer                     */
```

The **DosSearchPath** function searches the specified search path for the given filename. A search path is a null-terminated string that consists of a sequence of directory paths separated by semicolons (;). The function searches for the filename by looking in each directory (one directory at a time) in the order given.

Parameters

fsSearch Specifies how to interpret the *pszPath* parameter and whether or not to search the current directory. This parameter can be a combination of the following values:

Value	Meaning
SEARCH_CUR_DIRECTORY	The function searches the current directory before it searches the first directory in the search path. If this value is not specified, the function searches the current directory only if it is explicitly given in the search path.
SEARCH_ENVIRONMENT	The *pszPath* parameter points to the name of an environment variable. The function retrieves the value of the environment variable from the process's environment segment and uses it as the search path. If this value is not specified, *pszPath* points to a string that specifies the search path.
SEARCH_PATH	The *pszPath* parameter specifies the search path. This value cannot be used with the SEARCH_ENVIRONMENT value.

pszPath Points to a null-terminated string that specifies the search-path reference.

pszFileName Points to a null-terminated string that specifies the filename to search for. The string must be a valid MS OS/2 filename and can contain wildcard characters.

pbBuf Points to the buffer that receives the full pathname of the file if the filename is found.

cbBuf Specifies the length in bytes of the structure that is pointed to by the *pbBuf* parameter.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **DosSearchPath** function uses the search path pointed to by the *pszPath* parameter to look for the filename pointed to by the *pszFileName* parameter. The *pszPath* parameter can point to an environment variable name, such as PATH or DPATH, or it can point to a search path (as specified by the *fsSearch* parameter). The filename must be a valid MS OS/2 filename and can contain wildcard characters. If **DosSearchPath** finds a matching filename in any of the directories specified by the search path, the function copies the full, null-terminated pathname to the buffer pointed to by the *pbBuf* parameter. If the filename pointed to by the *pszFileName* parameter contains wildcard characters, the resulting pathname will also contain wildcard characters; the **DosFindFirst** function can be used to retrieve the actual filename(s).

The **DosSearchPath** function does not check for the validity of filenames. If the filename is not valid, the function returns an error indicating that the file was not found.

Example

This example uses the search path specified by the **DPATH** environment variable to search for the *abc.txt* filename:

```
CHAR szFoundFile[128];
DosSearchPath(SEARCH_ENVIRONMENT,   /* uses environment variable   */
    "DPATH",                        /* uses DPATH search path      */
    "abc.txt",                      /* filename                    */
    szFoundFile,                    /* receives resulting filename */
    sizeof(szFoundFile));           /* length of result buffer     */
```

The following example is identical to the first example if the **DPATH** variable is defined as shown:

```
DPATH=c:\sysdir;c:\init

DosSearchPath(SEARCH_PATH,           /* uses search path            */
    "c:\\sysdir;c:\\init",           /* search path                 */
    "abc.txt",                       /* filename                    */
    szFoundFile,                     /* receives resulting filename */
    sizeof(szFoundFile));            /* length of result buffer     */
```

See Also

DosFindFirst, DosScanEnv

DosSelectDisk

USHORT DosSelectDisk(*usDriveNumber* **)**
USHORT *usDriveNumber*; /* default-drive number */

The **DosSelectDisk** function selects the specified drive as the default drive for the calling process.

The **DosSelectDisk** function is a family API function.

Parameters

usDriveNumber Specifies the number of the default drive. Drive A is 1, drive B is 2, and so on.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_DRIVE

Example

This example calls the **DosSelectDisk** function to change the default drive to drive C. It then changes the default path to the root and opens the file *abc.txt*.

```
HFILE hf;
USHORT usAction;
DosSelectDisk(3);                /* selects drive C:              */
DosChDir("\\", 0L);              /* changes to the root directory */
DosOpen("abc.txt", &hf, &usAction, 0L, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE,
    OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYWRITE, 0L);
```

See Also

DosChDir, DosQCurDisk

■ DosSelectSession

USHORT DosSelectSession(*idSession***,** *ulReserved***)**
USHORT *idSession*; /* session identifier */
ULONG *ulReserved*; /* must be zero */

The **DosSelectSession** function switches the specified child session to the foreground. Only the parent session can call **DosSelectSession** to switch a session, and the parent session, or one of its descendant sessions, must be currently executing in the foreground when **DosSelectSession** is called. If the parent session is not in foreground, it can use **DosSelectSession** to switch itself to the foreground.

Parameters

idSession Specifies the identifier of the session to be switched to the foreground. This identifier must have been created previously by using the **DosStartSession** function. If *idSession* is 0x0000, the function switches the parent session to the foreground.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **DosSelectSession** function can only select a child session that was created by using the **DosStartSession** function with the **Related** field of the **STARTDATA** structure set to TRUE. In other words, this function cannot select sessions started as independent sessions.

See Also

DosSetSession, **DosStartSession**, **DosStopSession**

■ DosSemClear

USHORT DosSemClear(*hsem***)**
HSEM *hsem*; /* semaphore handle */

The **DosSemClear** function clears a system or RAM semaphore that has been set by using the **DosSemRequest**, **DosSemSet**, or **DosSemSetWait** function.

Parameters

hsem Identifies the semaphore to set. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_EXCL_SEM_ALREADY_OWNED
ERROR_INVALID_HANDLE

Comments

The **DosSemClear** function cannot clear a system semaphore that is owned by another process unless the semaphore is nonexclusive.

Example

This example uses the **DosSemClear** function to clear a RAM semaphore and a system semaphore:

```
ULONG hsem = 0;
HSYSSEM hsys;
DosSemClear(&hsem);          /* clears a RAM semaphore    */
DosSemClear(&hsys);          /* clears a system semaphore */
```

See Also

DosCreateSem, DosMuxSemWait, DosOpenSem, DosSemRequest, DosSemSet, DosSemWait

DosSemRequest

USHORT DosSemRequest(*hsem, lTimeOut***)**
HSEM *hsem*; /* semaphore handle */
LONG *lTimeOut*; /* time-out */

The **DosSemRequest** function requests that the specified semaphore be set as soon as it is clear. If no previous thread has set the semaphore, **DosSemRequest** sets the semaphore and returns immediately. If the semaphore has already been set by another thread, the function waits until a thread clears the semaphore (by using the **DosSemClear** function) or until a time-out occurs.

Parameters

hsem Identifies the semaphore to set. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

lTimeOut Specifies how long to wait for the semaphore to clear. If the value is greater then zero, this parameter specifies the number of milliseconds to wait before returning. If the value is SEM_IMMEDIATE_RETURN, the function returns immediately. If the value is SEM_INDEFINITE_WAIT, the function waits indefinitely.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INTERRUPT
ERROR_INVALID_HANDLE
ERROR_SEM_OWNER_DIED
ERROR_SEM_TIMEOUT
ERROR_TOO_MANY_SEM_REQUESTS

Comments

The effects of **DosSemRequest** are cumulative. If multiple calls to **DosSemRequest** set the semaphore, the same number of calls to the **DosSemClear** function are required to clear the semaphore.

If more than one thread has requested to set the semaphore, a thread may have to wait through several changes of the semaphore before it continues (depending on which thread clears the semaphore and when the system scheduler passes control to the thread). As long as the semaphore is set (even if it has been cleared and reset since the thread originally called the function), the thread must wait.

The **DosSemRequest** function cannot set a system semaphore that is set by another process, unless the semaphore is nonexclusive.

The **DosSemRequest** function can set system or RAM semaphores. A system semaphore is initially clear when it is created. A RAM semaphore is clear if its value is zero. Programs that use RAM semaphores should assign the initial value of zero.

Example

This example uses the **DosSemRequest** function to create a RAM semaphore. It also shows how to set and clear the semaphore.

```
ULONG hsem = 0;
DosSemRequest(&hsem,              /* address of handle    */
    -1L);                         /* waits indefinitely   */
    .
    .
    .
DosSemClear(&hsem);               /* clears the semaphore */
```

See Also

DosCreateSem, DosExitList, DosMuxSemWait, DosOpenSem, DosSemClear, DosSemSet, DosSemSetWait, DosSemWait

DosSemSet

USHORT DosSemSet(*hsem***)**
HSEM *hsem*; /* semaphore handle */

The **DosSemSet** function sets a specified semaphore. A process typically uses this function to set a semaphore, then waits for the semaphore to clear by using the **DosSemWait** or **DosMuxSemWait** function.

Parameters

hsem Identifies the semaphore to set. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_EXCL_SEM_ALREADY_OWNED
ERROR_INVALID_HANDLE
ERROR_TOO_MANY_SEM_REQUESTS

Comments

The **DosSemSet** function cannot set a system semaphore that is owned by another process unless the semaphore is nonexclusive.

Example

This example uses the **DosSemSet** function to set a RAM semaphore and a system semaphore:

```
ULONG hsem = 0;
HSYSSEM hsys;
DosSemSet(&hsem);         /* sets a RAM semaphore    */
DosSemSet(hsys);          /* sets a system semaphore */
```

See Also

DosCreateSem, DosMuxSemWait, DosOpenSem, DosSemClear, DosSemRequest, DosSemSetWait, DosSemWait

DosSemSetWait

```
USHORT DosSemSetWait(hsem, lTimeOut)
HSEM hsem;          /* semaphore handle */
LONG lTimeOut;      /* time-out         */
```

The **DosSemSetWait** function sets the specified semaphore (if it is not already set) and then waits for another thread to clear the semaphore (by using the **DosSemClear** function) or for a time-out to occur. The only difference between the **DosSemSetWait** function and the **DosSemWait** function is that the **DosSemSetWait** function will first set the semaphore if it is not already set.

Parameters

hsem Identifies the semaphore to set. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

lTimeOut Specifies how long to wait for the semaphore to become clear. If the value is greater then zero, this parameter specifies the number of milliseconds to wait before returning. If it is SEM_IMMEDIATE_RETURN, the function returns immediately. If it is SEM_INDEFINITE_WAIT, the function waits indefinitely.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_EXCL_SEM_ALREADY_OWNED
ERROR_INTERRUPT
ERROR_INVALID_HANDLE
ERROR_SEM_TIMEOUT
ERROR_TOO_MANY_SEM_REQUESTS

Comments

If more than one thread is setting and clearing the semaphore, a thread may have to wait through several changes of the semaphore before it can continue (depending on which thread clears the semaphore and when the system scheduler passes control to the thread). As long as the semaphore is set (even if it has been cleared and reset since the thread originally called the function), the thread must wait.

The **DosSemSetWait** function cannot be used to wait for a system semaphore that is set by another process unless the semaphore is nonexclusive.

Example

This example calls **DosSemSetWait** to set the specified RAM semaphore and then waits until another thread clears the semaphore. It waits for up to 5 seconds and then returns an ERROR_SEM_TIMEOUT error value if a time-out occurs before the semaphore is cleared.

```
#define INCL_DOSERRORS       /* include error constants */

ULONG hsem = 0;
if (DosSemSetWait(&hsem, 5000L) == ERROR_SEM_TIMEOUT) {
    .
    . /* error processing */
    .
}
else {
```

See Also

DosCreateSem, DosMuxSemWait, DosOpenSem, DosSemClear, DosSemRequest, DosSemWait

DosSemWait

```
USHORT DosSemWait(hsem, lTimeOut)
HSEM hsem;        /* semaphore handle */
LONG lTimeOut;    /* time-out         */
```

The **DosSemWait** function waits for a specified semaphore to be cleared. **DosSemWait** waits until a thread uses the **DosSemClear** function to clear the semaphore or until a time-out occurs. If no previous thread has set the semaphore, **DosSemWait** returns immediately.

Parameters

hsem Identifies the semaphore to set. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

lTimeOut Specifies how long to wait for the semaphore to clear. If the value is greater then zero, this parameter specifies the number of milliseconds to wait before returning. If the value is SEM_IMMEDIATE_RETURN, the function returns immediately. If the value is SEM_INDEFINITE_WAIT, the function waits indefinitely.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_EXCL_SEM_ALREADY_OWNED
ERROR_INTERRUPT
ERROR_INVALID_HANDLE
ERROR_SEM_TIMEOUT

Comments

The **DosSemWait** function cannot be used to wait for a system semaphore that is owned by another process unless the semaphore is nonexclusive.

If more than one thread is setting and clearing the semaphore, the thread calling **DosSemWait** may have to wait through several changes of the semaphore before it continues (depending on which thread clears the semaphore and when the system scheduler passes control to the calling thread). The thread must wait for as long as the semaphore is set, even if the semaphore has been cleared and reset since the thread originally called the function.

Example

This example calls the **DosSemWait** function to wait for up to 5 seconds for a RAM semaphore. If a time-out occurs before the semaphore handle is retrieved, the function returns an ERROR_SEM_TIMEOUT error value.

```
ULONG hsem = 0;
if (DosSemWait(&hsem, 5000L) == ERROR_SEM_TIMEOUT) {
    .
    . /* error processing */
    .
}
else {
```

See Also

DosCreateSem, DosMuxSemWait, DosOpenSem, DosSemRequest, DosSemSetWait, WinMsgSemWait

DosSendSignal

```
USHORT DosSendSignal(idProcess, usSigNumber)
USHORT idProcess;        /* process identifier of subtree root */
USHORT usSigNumber;      /* signal to send                     */
```

The **DosSendSignal** function sends a CTRL+C or CTRL+BREAK signal to the last descendant process that has a corresponding signal handler installed.

Parameters

idProcess Specifies the process identification code (PID) of the root process of the subtree. It is not necessary that this process still be running, but it is necessary that this process be a direct child of the process that issues this call.

usSigNumber Specifies the signal to send. It can be SIG_CTRLC to send a CTRL+C signal, or SIG_CTRLBREAK to send a CTRL+BREAK signal.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

DosFlagProcess, DosHoldSignal, DosSetSigHandler

DosSetCp

```
USHORT DosSetCp(usCodePage, usReserved)
USHORT usCodePage;     /* code-page identifier */
USHORT usReserved;     /* must be zero         */
```

The **DosSetCp** function sets the code-page identifier for the current process. The code-page identifier defines which translation table the system should use to translate input from the keyboard or to translate output to the screen and printer.

Parameters

usCodePage Specifies the code-page identifier.

usReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The file system activates the current code page for printer output whenever the printer is opened.

The code-page identifier can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

See Also

DosGetCp

DosSetDateTime

USHORT DosSetDateTime(*pdateTime*)
PDATETIME *pdateTime*; /* pointer to structure for date and time */

The **DosSetDateTime** function sets the current date and time. Although MS OS/2 maintains the current date and time, any process can change the date and time by using the **DosSetDateTime** function.

The **DosSetDateTime** function is a family API function.

Parameters

pdateTime Points to the structure that contains the date and time information. The **DATETIME** structure has the following form:

```
typedef struct _DATETIME {
    UCHAR   hours;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    SHORT   timezone;
    UCHAR   weekday;
} DATETIME;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_TS_DATETIME

Example

This example retrieves the current date and time and then calls the **DosSetDateTime** function to change the month to September and the day to the 26th:

```
DATETIME dateTime;
DosGetDateTime(&dateTime);      /* gets the current date and time */
dateTime.month = 9;             /* changes the month              */
dateTime.day = 26;              /* changes the day                */
DosSetDateTime(&dateTime);      /* sets the new date and time     */
```

See Also

DosGetDateTime

DosSetFHandState

USHORT DosSetFHandState(*hf*, *fsState*)
HFILE *hf*; /* file handle */
USHORT *fsState*; /* file-state flags */

The **DosSetFHandState** function modifies a file's inheritance, fail-on-error, and write-through. These flags are originally set by using the **DosOpen** function when the file is opened.

The **DosSetFHandState** function is a family API function.

Parameters

hf Identifies the handle of the file to be set. This handle must have been created previously by using the **DosOpen** function.

fsState Specifies the state of the file-handle. This parameter can be one or more of the following values:

Value	Meaning
OPEN_FLAGS_FAIL_ON_ERROR	Any function that uses the file handle returns immediately with an error value if there is an I/O error—for example, if the drive door is open or a sector is missing. If this value is not specified, the system passes the error to the system critical-error handler, which then reports the error to the user with a hard-error popup. The fail-on-error flag is not inherited by child processes. The fail-on-error flag applies to all functions that use the file handle, with the exception of the **DosDevIOCtl** function.
OPEN_FLAGS_NOINHERIT	The file handle is not available to any child process started by the current process. If this value is not specified, any child process started by the current process can use the file handle.
OPEN_FLAGS_WRITE_THROUGH	This flag applies to functions, such as **DosWrite**, that write data to the file. If this value is specified, the system writes data to the device before the given function returns. Otherwise, the system may store the data in an internal file buffer and write the data to the device only when the buffer is full or the file is closed.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER

Restrictions

In real mode, the following restriction applies to the **DosSetFHandState** function:

- Only the OPEN_FLAGS_NOINHERIT flag can be set.

Example

This example opens the file *abc* with the inheritance flag set to zero (all child processes inherit the file handle). It retrieves the current file-handle state, clears the bits that are required to be zero, sets the inheritance flag using the OR operator, and calls the **DosSetFHandState** function.

```
HFILE hf;
USHORT usAction, fState;
DosQFHandState(hf, &fState);                 /* gets the current state */
DosSetFHandState(hf,                         /* handle to the file     */
    (fState | OPEN_FLAGS_NOINHERIT));        /* set noinheritance flag */
```

See Also

DosBufReset, DosClose, DosDupHandle, DosExecPgm, DosOpen, DosQFHandState, DosSetMode, DosWrite

DosSetFileInfo

USHORT DosSetFileInfo (*hf*, *usInfoLevel*, *pfstsBuf*, *cbBuf*)

```
HFILE hf;               /* file handle                        */
USHORT usInfoLevel;     /* level of file information          */
PBYTE pfstsBuf;         /* pointer to file-status information */
USHORT cbBuf;           /* length of file-information buffer  */
```

The **DosSetFileInfo** function changes the time and date information for the specified file. The function replaces a file's time and date information with the information given in the structure pointed to by the *pfstsBuf* parameter.

The **DosSetFileInfo** function is a family API function.

Parameters

hf Identifies the file whose time and date information is being changed. This handle must have been created previously by using the **DosOpen** function.

usInfoLevel Specifies the level of file information being defined. In MS OS/2, version 1.1, this value must be 0x0001.

pfstsBuf Points to the **FILESTATUS** structure that contains the new information. The **FILESTATUS** structure has the following form:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

cbBuf Specifies the length in bytes of the structure pointed to by the *pfstsBuf* parameter.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_DIRECT_ACCESS_HANDLE
ERROR_INSUFFICIENT_BUFFER
ERROR_INVALID_FUNCTION
ERROR_INVALID_HANDLE
ERROR_INVALID_LEVEL

Comments

The **DosSetFileInfo** function does not change information in read-only files.

A zero in matching date and time fields will cause that aspect of file information to be left unchanged. For example, if both the **fdateCreation** and **ftimeCreation** fields are set to zero, both of these attributes are left unchanged.

See Also

DosNewSize, DosQFileInfo, DosSetFileMode

DosSetFileMode

```
USHORT DosSetFileMode(pszFileName, usAttribute, ulReserved)
PSZ pszFileName;        /* filename           */
USHORT usAttribute;     /* new file attribute */
ULONG ulReserved;       /* must be zero       */
```

The **DosSetFileMode** function sets the file attributes of the specified file. A file's mode is defined by the settings of its attributes.

The **DosSetFileMode** function is a family API function.

Parameters

pszFileName Points to a null-terminated string that specifies the name of the file. The string must be a valid MS OS/2 filename.

usAttribute Specifies the file's new attributes. This parameter can be a combination of the following values:

Value	Meaning
FILE_NORMAL	File can be read from or written to.
FILE_READONLY	File can be read from but not written to.
FILE_HIDDEN	File is hidden and does not appear when a directory is listed.
FILE_SYSTEM	File is a system file.
FILE_ARCHIVED	File has been archived.

The FILE_NORMAL value can be combined only with the FILE_ARCHIVED value.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION

See Also

DosQFileMode

DosSetFSInfo

```
USHORT DosSetFSInfo(usDriveNumber, usInfoLevel, pbBuf, cbBuf)
USHORT usDriveNumber;    /* drive number                                  */
USHORT usInfoLevel;      /* level of file-system information              */
PBYTE pbBuf;             /* pointer to structure for file-system information */
USHORT cbBuf;            /* length of buffer for file-system information  */
```

The **DosSetFSInfo** function sets information for a file-system device.

The **DosSetFSInfo** function is a family API function.

Parameters

usDriveNumber Specifies the logical drive number. The *usDriveNumber* parameter must be a value from 0 through 26. The default drive is 0, drive A is 1, drive B is 2, and so on.

usInfoLevel Specifies the level of file information required. In MS OS/2, version 1.1, this value must be 0x0002.

pbBuf Points to the structure that receives the information. When the request is for level–2 file information, this parameter points to a structure that contains the volume-label information. The **VOLUMELABEL** structure has the following form:

```
typedef struct _VOLUMELABEL {
    BYTE cch;
    CHAR achVolLabel[12];
} VOLUMELABEL;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

cbBuf Specifies the length (in bytes) of the **VOLUMELABEL** structure pointed to by the *pbBuf* parameter.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_CANNOT_MAKE
ERROR_INSUFFICIENT_BUFFER
ERROR_INVALID_DRIVE
ERROR_INVALID_LEVEL
ERROR_INVALID_NAME
ERROR_LABEL_TOO_LONG

See Also

DosQCurDisk, DosQFSInfo

DosSetMaxFH

```
USHORT DosSetMaxFH(usHandles)
USHORT usHandles;    /* number of file handles */
```

The **DosSetMaxFH** function sets the maximum number of file handles for the current process. The number of available handles limits the number of files that can be opened at once. However, all handles are not always available for use by the process. When determining the required number of handles, add several for the dynamic-link modules (these modules use several handles) and three for the default system input/output handles.

Parameters

usHandles Specifies the maximum number of file handles provided to the calling process. The maximum value for this parameter is 255; the default is 20.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Comments

This function preserves all currently open file handles.

See Also

DosDupHandle, DosOpen

■ DosSetNmPHandState

```
USHORT DosSetNmPHandState(hp, fsState)
HPIPE hp;           /* pipe handle */
USHORT fsState;     /* state flag  */
```

The **DosSetNmPHandState** function is used to set the read mode and the blocking mode of a named pipe.

Parameters

hp Identifies the pipe to read from.

fsState Specifies the new mode. The mode is a combination of a read-mode flag and a wait flag. The possible values are:

Value	Meaning
PIPE_READMODE_BYTE	Read pipe as a byte stream.
PIPE_READMODE_MESSAGE	Read pipe as a message stream.
PIPE_NOWAIT	Reading from and writing to the pipe returns immediately if no data is available.
PIPE_WAIT	Reading from and writing to the pipe waits if no data is available.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_INVALID_PARAMETER
ERROR_PIPE_NOT_CONNECTED

See Also

DosQNmPHandState

DosSetNmPipeSem

```
USHORT DosSetNmPipeSem(hp, hsem, usKeyVal)
HPIPE hp;             /* pipe handle         */
HSEM hsem;            /* semaphore handle    */
USHORT usKeyVal;      /* key value to associate */
```

The **DosSetNmPipeSem** function associates a semaphore with a named pipe.

Parameters

hp Identifies the named pipe.

hsem Identifies the semaphore to associate with the pipe.

usKeyVal Specifies a key identifier to associate with the named pipe.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_FUNCTION
ERROR_PIPE_NOT_CONNECTED
ERROR_SEM_NOT_FOUND

Comments

Up to two semaphores can be attached to a named pipe; one for the serving end of the pipe and one for the client end of the pipe. If a semaphore is already attached to one end of the named pipe, the old semaphore will be overwritten.

The **DosSetNmPipeSem** function only returns successfully for local named pipes. If the **DosSetNmPipeSem** function attempts to associate a semaphore with a remote named pipe, an ERROR_INVALID_FUNCTION error value is returned by the **DosSetNmPipeSem** function.

The **DosSetNmPipeSem** function allows a serving application that needs to handle a large number of incoming named pipes to avoid dedicating a thread for each named pipe and avoid polling the pipes. An application can instead call the **DosSemWait** or **DosMuxSemWait** function to determine when I/O can be performed on the pipe semaphore(s). This allows a large number of named pipes to be handled in an event-driven way, using only a small number of threads. The **DosQNmPipeSemState** function can be used to provide additional information about what I/O can be performed on the set of pipes.

See Also

DosMuxSemWait, DosQNmPipeSemState, DosSemWait

DosSetProcCp

```
USHORT DosSetProcCp(usCodePage, usReserved)
USHORT usCodePage;    /* code-page identifier */
USHORT usReserved;    /* must be zero         */
```

The **DosSetProcCp** function allows a process to set its code page.

Parameters

usCodePage Specifies a code-page-identifier word that has one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

usReserved Specifies a reserved value; must be zero.

Comments

This function sets the process code page of the calling process. The code page of a process is used in three ways. First, the printer code page is set to the process code page through the file system and Printer spooler (the system spooler must be installed) when the process makes a request to open the printer. Calling **DosSetProcCp** does not affect the code page of a printer opened before the call, nor does it affect the code page of a printer opened by another process. Second, country-dependent information will, by default, be retrieved encoded in the code page of the calling process. Third, a newly created process inherits its process code page from its parent process.

DosSetProcCp does not affect the screen or keyboard code page.

See Also

DosSetCp

DosSetPrty

```
USHORT DosSetPrty(fScope, fPrtyClass, sChange, id)
USHORT fScope;          /* indicates the scope of change */
USHORT fPrtyClass;      /* priority class to set         */
SHORT sChange;          /* change in priority level      */
USHORT id;              /* process or thread identifier  */
```

The **DosSetPrty** function sets the scheduling priority of the specified process or thread by changing the priority class and/or the priority level.

Within each class, a thread's priority level may vary—either through system action or through the **DosSetPrty** function. The system changes a thread's priority levels based on that thread's actions and the overall system activity.

Parameters

fScope Specifies the scope of the request. This parameter can be one of the following values:

Value	Meaning
PRTYS_PROCESS	Priority for the process and all its threads.
PRTYS_PROCESSTREE	Priority for the process and all child processes.
PRTYS_THREAD	Priority for one thread in the current process.

fPrtyClass Specifies the priority class of a process or thread. This parameter can be one of the following values:

Value	Meaning
PRTYC_IDLETIME	Idle-time.
PRTYC_NOCHANGE	No change; leave as is.
PRTYC_REGULAR	Regular.
PRTYC_TIMECRITICAL	Time-critical.

sChange Specifies the relative change in the current priority level of the process or thread. This parameter can be any value from -31 through +31, or the constants PRTYD_MINIMUM or PRTYD_MAXIMUM, which specify the minimum and maximum change allowed.

id Specifies a process or thread identifier, depending on the value of the *fScope* parameter.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_PCLASS
ERROR_INVALID_PDELTA
ERROR_INVALID_PROCID
ERROR_INVALID_SCOPE
ERROR_INVALID_THREADID
ERROR_NOT_DESCENDANT

See Also

DosEnterCritSec, DosGetInfoSeg, DosGetPrty

DosSetSession

```
USHORT DosSetSession(idSession, pstsdata)
USHORT idSession;          /* session identifier                          */
PSTATUSDATA pstsdata;      /* prior to structure for session-status data */
```

The **DosSetSession** function sets the status of a child session.

Parameters

idSession Specifies the identifier of the session for which the status is set. This identifier must have been created previously by using the **DosStartSession** function.

pstsdata Points to a **STATUSDATA** structure that contains the session-status data. The **STATUSDATA** structure has the following form:

```
typedef struct _STATUSDATA {
    USHORT Length;
    USHORT SelectInd;
    USHORT BindInd;
} STATUSDATA;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **DosSetSession** function allows a parent session to use the *SelectInd* and *BindInd* fields of the **STATUSDATA** structure to specify whether the child session can be selected by the user and whether or not the child session will also be brought to the foreground when the user brings the parent session to the foreground. These fields affect selections made by the user from the switch list; they do not affect selections made by the parent session. Each of these fields can be set individually, without affecting the current setting of the other.

A parent session can call the **DosSetSession** function only for a child session; neither the parent session itself nor any second-level child session can be set by using this function. The **DosSetSession** function can change the status of a child session only if that child was started as a related session; **DosSetSession** cannot change the status of sessions that were started as independent sessions.

A bond between a parent session and a child session can be broken by calling the **DosSetSession** function and specifying either **BindInd** = 2 to break the bond, or **BindInd** = 1 to break the bond and establish a new bond with a different child session.

A child session that is bound to a parent session will be brought to the foreground when the user selects the parent session, even if the status of the child session is nonselectable. If there is a bond between a parent session and a child session, and another bond between that child and a second-level child session, the second-level child session will be brought to the foreground when the user selects the parent session.

A parent session may be running in either the foreground or the background when **DosSetSession** is called.

The **DosSetSession** function may be called only by the process that started the session identified by the *idSession* parameter.

See Also

DosSelectSession, DosStartSession, DosStopSession

DosSetSigHandler

```
USHORT DosSetSigHandler(pfnSigHandler, pfnPrev, pfAction, fAction, usSigNumber)
PFNSIGHANDLER pfnSigHandler;       /* pointer to signal-handler function            */
PFNSIGHANDLER FAR * pfnPrev;       /* pointer to previous handler address           */
PUSHORT pfAction;                  /* pointer to variable for previous handler action */
USHORT fAction;                    /* type of request                               */
USHORT usSigNumber;                /* signal number                                 */
```

The **DosSetSigHandler** function installs or removes a signal handler for a specified signal. This function can also be used to ignore a signal or install a default action for a signal.

The **DosSetSigHandler** function is a family API function.

Parameters

pfnSigHandler Points to the address of the signal-handler function that receives control when a given signal occurs. For a full description, see the following "Comments" section.

pfnPrev Points to the variable that receives the address of the previous signal handler.

pfAction Points to the variable that receives the value of the previous signal handler's *fAction* parameter. The *pfAction* parameter can be a value from 0 through 3.

fAction Specifies the type of request. This parameter can be one of the following values:

Value	Meaning
SIGA_ACCEPT	The signal handler specified in the *pfnSigHandler* parameter will accept the signal specified in the *usSigNumber* parameter.
SIGA_ACKNOWLEDGE	The signal specified in the *usSigNumber* parameter is acknowledged. The signal handler specified in the *pfnSigHandler* parameter will accept the signal.
SIGA_ERROR	It is an error for any other process to signal this process with the signal specified in the *usSigNumber* parameter.
SIGA_IGNORE	Ignore the signal.
SIGA_KILL	Remove the signal handler.

usSigNumber Specifies the signal number. This parameter can be one of the following values:

Value	Meaning
SIG_BROKENPIPE	Connection to a pipe was broken.
SIG_CTRLBREAK	CTRL+BREAK.
SIG_CTRLC	CTRL+C.
SIG_KILLPROCESS	Program terminated.
SIG_PFLG_A	Process flag A.
SIG_PFLG_B	Process flag B.
SIG_PFLG_C	Process flag C.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_INVALID_FUNCTION
ERROR_INVALID_SIGNAL_NUMBER

Comments

The **DosSetSigHandler** function installs the signal handler that the system will call whenever the corresponding signal occurs. The signal handler is a function that responds to a signal by carrying out tasks (such as cleaning up files). A signal is an action initiated by the user or another process that temporarily suspends execution of the process while the signal is processed. Signals occur when the user presses the CTRL+C or CTRL+BREAK key sequences, when the process ends, or when another process calls the **DosFlagProcess** function. By default, the CTRL+C, CTRL+BREAK, and end-of-process signals terminate the process.

The signal-handler function can use the address and *fAction* parameter value of the previous signal handler to pass the signal through a chain of previous signal handlers. The new signal handler can also use the previous address and *fAction* value to restore the previous handler.

The **DosSetSigHandler** function acknowledges a signal and reenables it for subsequent input if the *fAction* parameter is set to SIGA_ACKNOWLEDGE. A process must acknowledge the signal while processing it to permit the signal to be used again.

The signal handler has the following form:

```
VOID PASCAL FAR FuncName(usSigArg, usSigNum)
USHORT usSigArg;     /* furnished by DosFlagProcess if appropriate */
USHORT usSigNum;     /* signal number being processed              */
{
    .
    .
    .
    return;
}
```

Parameters	Description
usSigArg	Specifies the signal argument passed by the process that sends the process-flag signal.
usSigNum	Specifies the signal number. This parameter can be any of the values listed for the *usSigNumber* parameter of the **DosSetSigHandler** function.

When a signal occurs, the system calls the corresponding signal handler, which then carries out tasks, such as displaying a message and writing and closing files. The signal handler receives control under the first thread of a process (thread 1). The thread that was executing when the signal occurred waits for signal processing to be completed. The signal handler can use the **return** statement to return control and restore execution of the waiting thread or the **DosExit** function to terminate the process.

The signal handler is not suspended when the **DosEnterCritSec** function is called. If a signal occurs, the processing done by the signal handler must not interfere with the processing that is done by the thread calling the **DosEnterCritSec** function.

All registers other than **cs**, **ip**, **ss**, **sp**, and **flags** in assembly-language signal handlers contain the same values as when the signal was received. The signal handler may exit by executing a far return instruction; execution resumes where it was interrupted, and all registers are restored to their values at the time of the interruption.

Restrictions

In real mode, the following restriction applies to the **DosSetSigHandler** function:

- Only the signal-break (SIG_CTRLBREAK) and signal-interrupt (SIG_CTRLC) signals are available. **DosSetSigHandler** may be used to install signal handlers for only these two signals.

See Also

DosCreateThread, DosFlagProcess, DosHoldSignal

DosSetVec

```
USHORT DosSetVec(usVecNum, pfnFunction, ppfnPrev)
USHORT usVecNum;     /* type of exception                          */
PFN pfnFunction;     /* pointer to function                        */
PPFN ppfnPrev;       /* pointer to variable for previous function's address */
```

The **DosSetVec** function installs or removes an exception handler for a specified exception. An exception is a program error, such as division by zero, that causes the system to pass control to the exception handler. The exception handler is an assembly-language routine that corrects errors or cleans up programs before terminating. The system calls the exception handler whenever the specified exception occurs. If a process does not install its own exception handler, the default exception handler terminates the process when an exception occurs.

The **DosSetVec** function is a family API function.

Parameters

usVecNum Specifies the number of the exception vector. This parameter can be one of the following values:

Value	Meaning
VECTOR_DIVIDE_BY_ZERO	Division by zero
VECTOR_EXTENSION_ERROR	Processor extension error
VECTOR_INVALIDOPCODE	Invalid opcode
VECTOR_NO_EXTENSION	Processor extension not available
VECTOR_OUTOFBOUNDS	Out of bounds
VECTOR_OVERFLOW	Overflow

pfnFunction Points to the address of the exception handler that receives control when the specified exception occurs. If this parameter is zero, the **DosSetVec** function removes the current exception handler. For a full description, see the following "Comments" section.

ppfnPrev Points to the variable that receives the address of the previous exception handler. The new exception handler can use this address to chain exception handling through all previous handlers or to restore the previous exception handler.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_FUNCTION

Comments

When the system calls the exception handler, interrupts are enabled and the machine status word and far return address are pushed on the stack. If the exception handler returns, it must use the **iret** (return from interrupt) instruction.

If the **DosSetVec** function is used to install an exception handler for vector VECTOR_EXTENSION_ERROR (processor extension not available), the function sets the machine status word (MSW) to indicate that no 80287 is available.

The emulate bit is set and the monitor processor bit is cleared. (This is done without regard for the true state of the hardware.) If the **DosSetVec** function is used to remove the exception handler for VECTOR_EXTENSION_ERROR, the function sets the machine status word to reflect the true state of the hardware.

Restrictions

In real mode, the following restriction applies to the **DosSetVec** function:

- Since the 8086 amd 8088 microprocessors do not raise this exception, *usVecNum* may not be VECTOR_EXTENSION_ERROR.

See Also

DosDevConfig, DosError

DosSetVerify

```
USHORT DosSetVerify(fVerify)
USHORT fVerify;    /* verify on/off */
```

The **DosSetVerify** function enables or disables data verification. When verification is enabled, the system verifies that data is written correctly whenever a process writes to a disk file.

The **DosSetVerify** function is a family API function.

Parameters

fVerify Specifies whether data verification is enabled. If the *fVerify* parameter is TRUE, verification is enabled. If it is FALSE, verification is disabled.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_VERIFY_SWITCH

Comments

Errors when writing to a disk file are very rare. This **DosSetVerify** function allows a process to verify the proper recording of critical data.

See Also

DosQVerify

DosSizeSeg

```
USHORT DosSizeSeg(sel, pulSize)
SEL sel;           /* segment selector     */
PULONG pulSize;    /* receives segment size */
```

The **DosSizeSeg** function retrieves the size (in bytes) of a specified segment.

The **DosSizeSeg** function is a family API function.

Parameters

sel Specifies the selector of the segment. For huge segments, this must be the base selector.

pulSize Points to the variable that receives the segment size (in bytes). (For huge segments, the number of full segments will be in the high word, and the size of the last segment will be in the low word. These values are equivalent to the values of the *usNumSeg* and *usPartialSeg* parameters that were passed to the **DosAllocHuge** or **DosReallocHuge** function.)

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

See Also **DosAllocHuge, DosAllocSeg, DosReallocHuge**

DosSleep

USHORT DosSleep(*ulTime* **)**
ULONG *ulTime*; /* number of milliseconds to wait */

The **DosSleep** function causes the current thread to wait for a specified interval or, if the specified interval is zero, to give up the remainder of the current time slice.

The **DosSleep** function is a family API function.

Parameters *ulTime* Specifies the number of milliseconds that the thread waits. This value is rounded up to the next clock tick.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_TS_WAKEUP

Comments The time the thread waits can be different from the specified time by a clock tick or two, depending on the execution status of the other threads running in the system. If the specified interval is zero, the process forgoes the remainder of its CPU time slice but is scheduled normally for its next time slice. When a process continues after suspension, its scheduled execution time could be delayed by hardware interrupts or by another thread running at a higher priority. If the time interval is not zero it is given in milliseconds, which are rounded up to the resolution of the scheduler clock. The **DosSleep** function should not be substituted for a real-time clock, because the rounding of the wait interval will cause inaccuracies to accumulate.

Example This example sets up a loop that waits for one second and then retrieves the time and date:

```
DATETIME date;
for (;;) {
    DosSleep(1000L);           /* waits for one second       */
    DosGetDateTime(&date);     /* retrieves the time and date */
    .
    .
    .
}
```

See Also **DosGetInfoSeg, DosTimerAsync, DosTimerStart**

DosStartSession

USHORT DosStartSession(*pstdata, pidSession, ppid***)**

```
PSTARTDATA pstdata;     /* pointer to structure containing session data */
PUSHORT pidSession;     /* pointer to variable for session identifier   */
PUSHORT ppid;           /* pointer to variable for process identifier   */
```

The **DosStartSession** function starts a session (screen group) and specifies the name of the program to start in that session. This function creates either an independent session or a child session, depending on the value of the **Related** field in the **STARTDATA** structure.

Parameters

pstdata Points to the **STARTDATA** structure that contains data describing the session to start. The **STARTDATA** structure has the following form:

```
typedef struct _STARTDATA {
    USHORT Length;
    USHORT Related;
    USHORT FgBg;
    USHORT TraceOpt;
    PSZ    PgmTitle;
    PSZ    PgmName;
    PBYTE  PgmInputs;
    PBYTE  TermQ;
    PBYTE  Environment;
    USHORT InheritOpt;
    USHORT SessionType;
    PSZ    IconFile;
    ULONG  PgmHandle;
    USHORT PgmControl;
    USHORT InitXPos;
    USHORT InitYPos;
    USHORT InitXSize;
    USHORT InitYSize;
} STARTDATA;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pidSession Points to the variable that receives the identifier of the child session.

ppid Points to the variable that receives the process identifier of the child process.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The MS OS/2 session manager writes a data element into the specified queue when the child session created by the **DosStartSession** function terminates. A parent session can be notified when a child session has terminated by using the **DosReadQueue** function. When the child session terminates, the request value returned by **DosReadQueue** is zero, and the data-element format consists of two unsigned values: the session identifier and the result code.

Only the process that calls the **DosStartSession** function should call the **DosReadQueue** function. Only this process can address the notification data element. After reading and processing the data element, the calling process must use the **DosFreeSeg** function to free the segment that contains the data element.

A child session is created when the **Related** field of the **STARTDATA** structure is set to TRUE. The process identifier of the child process cannot be used with MS OS/2 functions, such as **DosSetPrty**, that require a parent process/child process relationship.

An independent session is created when the **Related** field of the **STARTDATA** structure is set to TRUE. An independent session is not under the control of the starting session. The **DosStartSession** function does not copy session and process identifiers for an independent session to the *pidSession* and *ppid* parameters.

New sessions can be started in the foreground only when the caller's session (or one of the caller's descendant sessions) is currently executing in the foreground. The new session appears in the shell switch list.

See Also

DosCreateQueue, DosExecPgm, DosFreeSeg, DosReadQueue, DosSelectSession, DosSetSession, DosStopSession

DosStopSession

```
USHORT DosStopSession(fScope, idSession, ulReserved)
USHORT fScope;       /* all sessions/specified session stopped */
USHORT idSession;    /* session identifier                     */
ULONG ulReserved;    /* must be zero                           */
```

The **DosStopSession** function terminates a session that was started by using the **DosStartSession** function.

Parameters

fScope Specifies whether the function stops all sessions or only the specified session. If the *fScope* parameter is 0x0000, the function stops only the specified session. If it is 0x0001, the function stops all sessions.

idSession Specifies the identifier of the session to be stopped. This identifier must have been created previously by using the **DosStartSession** function. This parameter is ignored if the *fScope* parameter is set to 0x0001.

ulReserved Specifies a reserved value; must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **DosStopSession** function can terminate only child sessions that were started by using the **DosStartSession** function (child sessions of the terminated session will terminate as well). Sessions that were started as independent sessions cannot be terminated by using **DosStopSession**.

A parent session can be running in either the foreground or the background when **DosStopSession** is issued. If a child session is in the foreground when it is stopped, the parent session becomes the foreground session. The **DosStopSession** function breaks any bond between the parent session and the specified child session.

A process running in the session specified by the *idSession* parameter can refuse to terminate. If this happens, **DosStopSession** returns zero. To verify that the target session has terminated, a process can wait for notification through the termination queue that is specified in the **DosStartSession** function.

See Also

DosSetSession, DosStartSession

DosSubAlloc

```
USHORT DosSubAlloc(sel, pusOffset, cbBlock)
SEL sel;              /* segment selector                   */
PUSHORT pusOffset;    /* pointer to variable for offset     */
USHORT cbBlock;       /* number of bytes of requested memory */
```

The **DosSubAlloc** function allocates memory in a segment that was allocated previously by using the **DosAllocSeg** or **DosAllocShrSeg** function and that was initialized by using the **DosSubSet** function.

The **DosSubAlloc** function is a family API function.

Parameters

sel Specifies the selector of the data segment in which the memory should be allocated.

pusOffset Points to the variable that receives the offset to the allocated block.

cbBlock Specifies the size (in bytes) of the requested memory block.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_DOSSUB_BADSIZE
ERROR_DOSSUB_NOMEM

Comments

The *cbBlock* parameter must not be greater than the maximum size of the segment minus 8 bytes. Since all memory blocks are aligned on byte boundaries, the *cbBlock* parameter does not need to be a multiple of 16.

See Also

DosAllocSeg, DosAllocShrSeg, DosSubFree, DosSubSet

DosSubFree

```
USHORT DosSubFree(sel, offBlock, cbBlock)
SEL sel;              /* segment selector                */
USHORT offBlock;      /* block offset                    */
USHORT cbBlock;       /* number of bytes in block to free */
```

The **DosSubFree** function frees memory that was allocated previously by using the **DosSubAlloc** function.

The **DosSubFree** function is a family API function.

Parameters

sel Specifies the selector of the data segment from which the memory should be freed.

offBlock Specifies the offset of the memory block to be freed. This offset must have been created previously by using the **DosSubAlloc** function.

cbBlock Specifies the size (in bytes) of the block to free.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_DOSSUB_BADSIZE
ERROR_DOSSUB_OVERLAP

See Also

DosAllocSeg, DosSubAlloc, DosSubSet

DosSubSet

```
USHORT DosSubSet(sel, fFlags, cbSeg)
SEL sel;            /* segment selector                    */
USHORT fFlags;      /* initialize/increase size of segment */
USHORT cbSeg;       /* new size of block                   */
```

The **DosSubSet** function initializes a segment for suballocation or changes the size of a previously initialized segment.

The **DosSubSet** function is a family API function.

Parameters

sel Specifies the selector of the data segment.

fFlags Specifies whether to initialize the segment or increase its size. If the *fFlags* parameter is 0x0001, the function initializes the segment. If *fFlags* is 0x0000, the function changes the size of the segment.

cbSeg Specifies the new size (in bytes) of the segment.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_DOSSUB_BADFLAG
ERROR_DOSSUB_BADSIZE
ERROR_DOSSUB_SHRINK

Comments

If the *fFlags* parameter is 0x0001, the **DosSubSet** function initializes the segment so that the **DosSubAlloc** function can be used to allocate memory blocks in the segment. The segment must have been allocated previously by using the **DosAllocSeg** or **DosAllocShrSeg** function.

If the *fFlags* parameter is 0x0000, the **DosSubSet** function changes the size of the segment to the number of bytes specified by the *cbSeg* parameter. If the specified size is greater than the current size of the segment, the **DosReallocSeg** function must be called before **DosSubSet**. If **DosSubSet** is not called after changing the size of a segment by using **DosReallocSeg**, the results can be unpredictable.

When changing the size of a segment by using the **DosSubSet** function, the *cbSeg* parameter must be a multiple of 4 bytes that is greater than or equal to 12 bytes, or it must be zero. Otherwise, the size is rounded up to the next multiple of 4. In the **DosSubSet** function, setting the *cbSeg* parameter to zero indicates that the segment is 64K, but in the **DosSubAlloc** and **DosSubFree** functions, it is an error when the *cbSeg* parameter is equal to zero.

See Also

DosAllocSeg, DosAllocShrSeg, DosReallocSeg, DosSubAlloc, DosSubFree

DosSuspendThread

```
USHORT DosSuspendThread(tid)
TID tid;    /* identifier of thread to suspend */
```

The **DosSuspendThread** function suspends the execution of a thread until a call to the **DosResumeThread** function is made that specifies the suspended thread's identifier.

Parameters

tid Specifies the thread identifier of the thread to be suspended.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_INVALID_THREADID

Comments

The specified thread may not be suspended immediately if it has called a system function that has locked some system resources; the locked resources must be freed before the thread is suspended. The thread will not continue to execute until a **DosResumeThread** function is called.

A thread can suspend threads only within its process.

See Also

DosCreateThread, DosEnterCritSec, DosResumeThread

DosTimerAsync

```
USHORT DosTimerAsync(ulTime, hsem, phtimer)
ULONG ulTime;       /* time before semaphore is cleared  */
HSEM hsem;          /* system-semaphore handle           */
PHTIMER phtimer;    /* pointer to variable for timer handle */
```

The **DosTimerAsync** function creates a timer that counts for a specified number of milliseconds, then clears a specified semaphore.

Parameters

ulTime Specifies the time (in milliseconds) before the semaphore is cleared. This value is rounded up to the next clock tick, if necessary.

hsem Identifies the system semaphore that signals the end of the timer. This handle must have been created previously by using the **DosCreateSem** function.

phtimer Points to the variable that receives the timer handle.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_TS_NOTIMER
ERROR_TS_SEMHANDLE

Comments

The timer runs asynchronously—that is, while the timer counts the time, the **DosTimerAsync** function returns to let the process continue to execute other tasks. The timer counts the time only once.

The given semaphore must be a system semaphore. If the process uses the semaphore to determine when data is available, it must use the **DosSemSet** function to set the semaphore before calling the **DosTimerAsync** function.

The timer handle specified by the *phtimer* parameter can be used by the **DosTimerStop** function to cancel the timer.

The **DosTimerAsync** function is similar to the **DosSleep** function except that **DosTimerAsync** returns immediately; **DosSleep** returns only after the specified time has elapsed.

See Also **DosSemSet, DosSleep, DosTimerStart, DosTimerStop**

DosTimerStart

```
USHORT DosTimerStart(ulTime, hsem, phtimer)
ULONG ulTime;          /* time before semaphore is cleared  */
HSEM hsem;             /* system-semaphore handle           */
PHTIMER phtimer;       /* pointer to variable for timer handle */
```

The **DosTimerStart** function creates a timer that counts for a specified number of milliseconds, then clears the specified semaphore. The function repeats this process continually, counting the time and clearing the semaphore, until the process stops it by using the **DosTimerStop** function. The timer handle is used in the **DosTimerStop** function to cancel the timer.

Parameters

ulTime Specifies the time (in milliseconds) before the semaphore is cleared.

hsem Identifies the system semaphore that signals the end of the timer. This handle must have been created previously by using the **DosCreateSem** function.

phtimer Points to the variable that receives the timer handle.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_TS_NOTIMER
ERROR_TS_SEMHANDLE

Comments

The timer runs asynchronously—that is, while the timer counts the time, the function returns to let the process continue to execute other tasks.

The given semaphore must be a system semaphore. If the process uses the semaphore to determine when data is available, it must use the **DosSemSet** function to set the semaphore before calling the **DosTimerStart** function.

If necessary, the **DosTimerStart** function rounds up the *ulTime* parameter to the next clock tick.

The timer may clear the semaphore several times before a process that is waiting for the semaphore resumes execution. If the process requires an accurate count of the time it waited, it should retrieve the current system time from the global information segment before and after waiting for the semaphore and compare these times.

See Also **DosGetInfoSeg, DosSemSet, DosTimerStop**

DosTimerStop

```
USHORT DosTimerStop(htimer)
HTIMER htimer;      /* timer handle */
```

The **DosTimerStop** function stops a specified timer.

Parameters

htimer Identifies the timer to be stopped. This handle must have been created previously by using the **DosTimerAsync** or **DosTimerStart** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_TS_HANDLE

Comments

When the **DosTimerStop** function stops a timer, it does not clear the corresponding semaphore. If a process is waiting for the semaphore to clear, the process that stops the timer should also clear the semaphore.

See Also

DosTimerAsync, DosTimerStart

DosTransactNmPipe

```
USHORT DosTransactNmPipe(hp, pbOutBuf, cbOutBuf, pbInBuf, cbInBuf, pcbRead)
HPIPE hp;           /* pipe handle                                  */
PBYTE pbOutBuf;     /* pointer to buffer containing data            */
USHORT cbOutBuf;    /* number of bytes in output buffer             */
PBYTE pbInBuf;      /* pointer to buffer receiving data             */
USHORT cbInBuf;     /* number of bytes in input buffer              */
PUSHORT pcbRead;    /* pointer to variable receiving number of bytes read */
```

The **DosTransactNmPipe** function writes data to and reads data from a named pipe.

Parameters

hp Identifies the named pipe.

pbOutBuf Points to the buffer containing the data that is written to the pipe.

cbOutBuf Specifies the size (in bytes) of the output buffer.

pbInBuf Points to the input buffer that receives the data read from the pipe.

cbInBuf Specifies the size (in bytes) of the input buffer.

pcbRead Points to the variable that receives the number of bytes read from the pipe.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_INTERRUPT
ERROR_INVALID_FUNCTION
ERROR_SEM_TIMEOUT

Comments

The **DosTransactNmPipe** function fails if the named pipe contains any unread data or if the named pipe is not in message mode. A named pipe's blocking state has no effect on the **DosTransactNmPipe** function. The **DosTransactNmPipe** function does not return until data is written into the output buffer.

See Also

DosCallNmPipe

■ DosUnlockSeg

```
USHORT DosUnlockSeg(sel)
SEL sel;    /* selector of segment to unlock */
```

The **DosUnlockSeg** function unlocks a discardable segment. Once a segment is unlocked, the system may discard it to make space available for other segments.

Parameters

sel Specifies the selector of the segment to unlock.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

DosUnlockSeg applies only to segments that are allocated by using the **Dos-AllocSeg** function with the *fsAlloc* parameter set to SEG_DISCARDABLE.

The **DosLockSeg** and **DosUnlockSeg** functions may be nested. If **DosLockSeg** is called 5 times to lock a segment, **DosUnlockSeg** must be called 5 times to unlock the segment. A segment becomes permanently locked if it is locked 255 times without being unlocked.

See Also

DosAllocSeg, DosLockSeg

■ DosWaitNmPipe

```
USHORT DosWaitNmPipe(pszName, ulTimeOut)
PSZ pszName;        /* pointer to pipe name */
ULONG ulTimeOut;    /* timeout value        */
```

The **DosWaitNmPipe** function waits for a named pipe to become available.

Parameters

pszName Points to the name of the pipe. The name is in the form **\pipe***name* for a local pipe and *****server***\pipe***name* for a remote pipe.

ulTimeOut Specifies a value (in milliseconds) that is the amount of time MS OS/2 should wait for the pipe to become available.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_BAD_PIPE
ERROR_INTERRUPT
ERROR_SEM_TIMEOUT

Comments

The **DosWaitNmPipe** function should be used only when the **DosOpen** function returns the ERROR_PIPE_BUSY error value.

If more than one process has requested a named pipe that has become available, the system gives the pipe to the process that has been waiting the longest.

See Also

DosOpen

■ DosWrite

```
USHORT DosWrite(hf, pvBuf, cbBuf, pcbBytesWritten)
HFILE hf;                    /* file handle                          */
PVOID pvBuf;                 /* pointer to buffer                    */
USHORT cbBuf;                /* number of bytes to write             */
PUSHORT pcbBytesWritten;     /* pointer to variable receiving byte count */
```

The **DosWrite** function writes data from a buffer to a file, then copies the number of bytes written to a variable.

The **DosWrite** function is a family API function.

Parameters

hf Identifies the file that receives the data. This handle must have been created previously by using the **DosOpen** function.

pvBuf Points to the buffer that contains the data to write.

cbBuf Specifies the number of bytes to write.

pcbBytesWritten Points to the variable receiving the number of bytes written.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NOT_DOS_DISK
ERROR_WRITE_FAULT

Comments

The **DosWrite** function begins to write at the current file-pointer position. The file-pointer position can be changed by using the **DosChgFilePtr** function.

If the specified file has been opened using the write-through flag, the **DosWrite** function writes data to the disk before returning. Otherwise, the system collects the data in an internal file buffer and writes the data to the disk only when the buffer is full.

The **DosWrite** function may write fewer bytes to the file than the number specified in the *cbBuf* parameter if there is not enough space on the disk for all of the requested bytes. The *cbBuf* parameter can be zero without causing an error—that is, writing no bytes is acceptable.

The efficiency with which the **DosWrite** function writes to a disk is improved when the *cbBuf* parameter is set to a multiple of the disk's bytes-per-sector size. When *cbBuf* is set this way, the function writes directly to the disk, without first copying the data to an internal file buffer. (The **DosQFSInfo** function retrieves the byters-per-sector value for a disk.)

Example

This example creates the file *abc* and calls the **DosWrite** function to write the contents of the *abBuf* buffer to the file:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesWritten, usError;
usError = DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL,
    FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, 0L);
if (!usError) {
    DosWrite(hf,                    /* file handle              */
        abBuf,                      /* buffer address           */
        sizeof(abBuf),              /* buffer size              */
        &cbBytesWritten);           /* address of bytes written */
```

See Also

DosChgFilePtr, DosOpen, DosRead, DosWriteAsync

DosWriteAsync

```
USHORT DosWriteAsync(hf, hsemRam, pusErrCode, pvBuf, cbBuf, pcbBytesWritten)
HFILE hf;                     /* file handle                              */
PULONG hsemRam;               /* pointer to RAM semaphore                 */
PUSHORT pusErrCode;           /* pointer to variable for error value      */
PVOID pvBuf;                  /* pointer to buffer containing data to write */
USHORT cbBuf;                 /* number of bytes in buffer                */
PUSHORT pcbBytesWritten;      /* pointer to variable for bytes written    */
```

The **DosWriteAsync** function writes one or more bytes of data to a specified file. The function writes the data asynchronously—that is, the function returns immediately, but continues to copy data to the specified file while the process continues with other tasks.

Parameters

hf Identifies the file that receives the data. This handle must have been created previously by using the **DosOpen** function.

hsemRam Points to the RAM semaphore that indicates when the function has finished reading the data.

pusErrCode Points to the variable that receives an error value.

pvBuf Points to the buffer that contains the data to write.

cbBuf Specifies the number of bytes to write.

pcbBytesWritten Points to the variable receiving the number of bytes written.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NO_PROC_SLOTS
ERROR_NOT_DOS_DISK
ERROR_WRITE_FAULT

Comments

The **DosWriteAsync** function starts writing at the current file-pointer position. The file-pointer position can be changed by using the **DosChgFilePtr** function.

If the specified file has been opened using the write-through flag, the **DosWriteAsync** function writes data to the disk as well as to the file before returning. If the write-through flag has not been set, the system collects the data in an internal file buffer and writes the data to the disk only when the buffer is full.

The **DosWriteAsync** function may write fewer bytes to the file than the number specified in the *cbBuf* parameter if there is not enough space on the disk for all of the requested bytes. The *cbBuf* parameter can be zero without causing an error—that is, writing no bytes is acceptable.

When the **DosWriteAsync** function has written the data, it clears the RAM semaphore pointed to by the *hsemRam* parameter. If the process uses the semaphore to determine when data is available, it must use the **DosSemSet** function to set the semaphore before calling **DosWriteAsync**.

The efficiency with which the **DosWriteAsync** function writes to a disk is improved when the *cbBuf* parameter is set to a multiple of the disk's bytes-per-sector size. When *cbBuf* is set this way, the function writes directly to the disk, without first copying the data to an internal file buffer. (The **DosQFSInfo** function retrieves the byters-per-sector value for a disk.)

Example

This example creates the file *abc.ext*, sets a RAM semaphore, and calls the **DosWriteAsync** function to write the contents of the buffer *abBuf* to a file. When any additional processing is finished, the example calls the **DosSemWait** function to wait until **DosWriteAsync** has finished writing to the file.

```
ULONG hsemWrite = 0;
BYTE abBuf[1024];
HFILE hf;
USHORT usAction, cbBytesWritten;
USHORT usWriteAsyncError;
DosOpen("abc.ext", &hf, &usAction, OL, FILE_NORMAL,
    FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, OL);

DosSemSet(&hsemWrite);          /* sets the semaphore        */
DosWriteAsync(hf,               /* file handle               */
    &hsemWrite,                 /* semaphore address         */
    &usWriteAsyncError,         /* return-code address       */
    abBuf,                      /* buffer address            */
    sizeof(abBuf),              /* buffer size               */
    &cbBytesWritten);           /* address of bytes written */
    .
    . /* Other processing would go here */
    .
DosSemWait(&hsemWrite, -1L);    /* waits for DosWriteAsync   */
if (usWriteAsyncError) {
    .
    . /* Error processing would go here. */
    .
```

See Also

DosChgFilePtr, DosOpen, DosQFSInfo, DosReadAsync, DosSemSet, DosSemWait, DosWrite

DosWriteQueue

```
USHORT DosWriteQueue(hqueue, usRequest, cbBuf, pbBuf, usPriority)
HQUEUE hqueue;          /* handle of target queue                   */
USHORT usRequest;       /* request/identification data              */
USHORT cbBuf;           /* number of bytes to write                 */
PBYTE pbBuf;            /* pointer to buffer containing element to write */
UCHAR usPriority;       /* priority of element to write             */
```

The **DosWriteQueue** function writes an element to the specified queue. The position of the element in the queue is determined by the value that was specified in the *fQueueOrder* parameter of the **DosCreateQueue** function when the queue was created; if the value of this parameter was set to 0x0002 (priority queue), the *usPriority* parameter of the **DosWriteQueue** function can be used to set the priority of the element. After the element is written, the process that owns the queue may read the element by using the **DosPeekQueue** or **DosReadQueue** function.

Parameters

hqueue Identifies the queue to be written to. This handle must have been previously created or opened by using the **DosCreateQueue** or **DosOpenQueue** function.

usRequest Specifies a program-supplied event code. MS OS/2 does not use this field; it is reserved for the program's use. The queue owner can retrieve this value by using the **DosPeekQueue** or **DosReadQueue** function.

cbBuf Specifies the number of bytes to be copied to the buffer that is pointed to by the *pbBuf* parameter.

pbBuf Points to the buffer that contains the element to be written to the queue.

usPriority Specifies the element priority. This parameter can be any value from 0 through 15; 15 is the highest priority.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_QUE_INVALID_HANDLE
ERROR_QUE_NO_MEMORY

Comments

The **DosWriteQueue** function returns an error value if the queue has been closed by the process that owns it.

If the queue owner uses a RAM semaphore to notify it when elements are added to the queue, the semaphore must be shared. If the notifying semaphore is a system semaphore, the writing process must have opened the semaphore by using the **DosOpenSem** function.

Example

This example creates a queue and calls the **DosWriteQueue** function to write the string "Hello World" to the queue:

```
HQUEUE hqueue;
DosCreateQueue(&hqueue, 0, "\\queues\\abc.que");
DosWriteQueue(hqueue,                       /* handle to queue   */
    0,                                      /* request data      */
    11,                                     /* length of data    */
    "Hello World",                          /* data buffer       */
    0);                                     /* element priority */
```

See Also

DosCreateQueue, DosOpenQueue, DosReadQueue

KbdCharIn

```
USHORT KbdCharIn(pkbci, fWait, hkbd)
PKBDKEYINFO pkbci;   /* pointer to structure for keystroke info. */
USHORT fWait;        /* wait/no-wait flag                        */
HKBD hkbd;           /* keyboard handle                          */
```

The **KbdCharIn** function retrieves character and scan-code information from a logical keyboard. The function copies the information to the structure pointed to by the *pkbci* parameter. Keystroke information includes the character value of a given key, the scan code, the keystroke status, the state of the shift keys, and the system time (in milliseconds) when the keystroke occurred. For information on scan codes, key codes, and MS OS/2 control and editing keys, see Chapter 5, "File Formats."

The **KbdCharIn** function is a family API function.

Parameters

pkbci Points to the **KBDKEYINFO** structure that receives the keystroke information. The **KBDKEYINFO** structure has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

fWait Specifies whether to wait for keystroke information if none is available. If this parameter is IO_WAIT, the function waits for a keystroke if one is not available. If the parameter is IO_NOWAIT, the function returns immediately whether or not it retrieved any keystroke information. The **fbStatus** field in the **KBDKEYINFO** structure specifies whether a keystroke is received. The **fbStatus** field is nonzero if a keystroke is received or zero if not.

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_KBD_FOCUS_REQUIRED
ERROR_KBD_INVALID_IOWAIT

Comments

The **KbdCharIn** function copies and removes keystroke information from the input buffer of the specified logical keyboard. Although echo mode for the logical keyboard may be turned on, **KbdCharIn** does not echo the characters it reads. If the keyboard is in ASCII mode, **KbdCharIn** retrieves keystroke information for each key pressed except shift keys and MS OS/2 CTRL keys. If the keyboard is in binary mode, **KbdCharIn** retrieves keystroke information for any key pressed except shift keys. In most cases, a shift key is pressed in combination with other keys to create a single keystroke. In binary mode with shift report turned on, a key by itself creates a keystroke that this function can retrieve. For more information on binary mode and shift-report mode, see the **KbdSetStatus** function.

The **KbdCharIn** function retrieves extended ASCII codes, such as when the ALT key and another key, called the primary key, are pressed simultaneously. When the function retrieves an extended code, it sets the **chChar** field of the **KBDKEYINFO** structure to 0x0000 or 0x00E0 and copies the extended code to the **chScan** field. The extended code is usually the scan code of the primary key. In ASCII mode, the function retrieves only complete extended codes, which means that if both bytes of the extended code do not fit in the buffer, neither byte is retrieved. For more information on extended ASCII codes, see Appendix C, "Country and Code-Page Information."

This function must be called twice to retrieve a code for a double-byte character set (DBCS). If the code retrieved is the first byte of a double-byte character, the **fbStatus** field of the **KBDKEYINFO** structure is set to 0x0080.

Restrictions

In real mode, the following restrictions apply to the **KbdCharIn** function:

- It does not copy the system time to the **KBDKEYINFO** structure and there is no interim character support.
- It retrieves characters only from the default logical keyboard (handle 0).
- The **fbStatus** field may be 0x0000 or SHIFT_KEY_IN.
- The *hkbd* parameter is ignored.

Example

This example calls the **KbdCharIn** function to retrieve a character, and then displays the character on the screen:

```
KBDKEYINFO kbci;
KbdCharIn(&kbci,                     /* structure for data */
    IO_WAIT,                         /* waits for key      */
    0);                              /* keyboard handle    */
VioWrtTTY(&kbci.chChar, 1, 0);
```

See Also

KbdGetStatus, KbdOpen, KbdPeek, KbdSetStatus, KbdStringIn

KbdClose

USHORT KbdClose(*hkbd* **)**
HKBD *hkbd*;   /* keyboard handle */

The **KbdClose** function closes the specified logical keyboard. The function removes any remaining keystrokes from the input buffer and automatically frees the focus (if the logical keyboard has it).

The default keyboard cannot be closed. If you specify the default keyboard (handle 0), the **KbdClose** function ignores the request.

Parameters

hkbd Identifies the logical keyboard to close. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_KBD_INVALID_HANDLE

Example This example opens a logical keyboard and calls **KbdClose** to close it:

```
HKBD hkbd;
KbdOpen(&hkbd);
    .
    .
    .
KbdClose(hkbd);
```

See Also **KbdFlushBuffer, KbdFreeFocus, KbdOpen**

■ KbdDeRegister

USHORT KbdDeRegister(*void* **)**

The **KbdDeRegister** function restores the default **Kbd** subsystem and releases any previously registered **Kbd** subsystem. The function restores the default **Kbd** subsystem for all processes in the current screen group.

Once a process registers a **Kbd** subsystem, no other process in the screen group may register a **Kbd** subsystem until the default subsystem is restored. Only the process registering a **Kbd** subsystem may call the **KbdDeRegister** function to restore the default subsystem.

Parameters This function has no parameters.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_KBD_DEREGISTER

See Also **KbdRegister**

■ KbdFlushBuffer

USHORT KbdFlushBuffer(*hkbd* **)**
HKBD *hkbd*; /* keyboard handle */

The **KbdFlushBuffer** function removes all keystroke information from the input buffer of the specified logical keyboard, but only if the keyboard has the focus or is the default keyboard.

The **KbdFlushBuffer** function is a family API function.

Parameters *hkbd* Identifies the logical keyboard to clear. The handle must have been created previously by using the **KbdOpen** function.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

Restrictions In real mode, the following restriction applies to the **KbdFlushBuffer** function:

- The *hkbd* parameter is ignored.

Example This example opens a logical keyboard and calls **KbdFlushBuffer** to remove any keystrokes from the input buffer:

```
HKBD hkbd;
KbdOpen(&hkbd);
    .
    .
    .
KbdFlushBuffer(hkbd);
```

See Also **KbdCharIn**

■ KbdFreeFocus

USHORT KbdFreeFocus(*hkbd***)**
HKBD *hkbd*; /* keyboard handle */

The **KbdFreeFocus** function frees the focus from the specified logical keyboard. Other logical keyboards can then use the focus.

Parameters *hkbd* Identifies the logical keyboard that loses the focus. The handle must have been created previously by using the **KbdOpen** function.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

Comments If a process has been waiting for the focus as a result of calling the **KbdGetFocus** function, MS OS/2 assigns the focus to the logical keyboard as soon as it is free. If more than one process is waiting, MS OS/2 chooses a logical keyboard and assigns the focus. The other processes continue to wait until the focus is free.

Example This example frees a logical keyboard: if other logical keyboards have been waiting, MS OS/2 assigns the focus to one of them; if no other logical keyboards have been waiting, MS OS/2 uses the default keyboard:

```
HKBD hkbd;
KbdOpen(&hkbd);
KbdGetFocus(IO_WAIT, hkbd);      /* gets focus  */
    .
    .
    .
KbdFreeFocus(hkbd);              /* frees focus */
```

See Also **KbdGetFocus, KbdOpen**

■ KbdGetCp

USHORT KbdGetCp(*ulReserved*, *pidCodePage*, *hkbd***)**
ULONG *ulReserved*; /* must be zero */
PUSHORT *pidCodePage*; /* pointer to code-page identifier */
HKBD *hkbd*; /* keyboard handle */

The **KbdGetCp** function retrieves the current code-page identifier for the specified logical keyboard. The code-page identifier defines which translation table MS OS/2 uses to translate keystrokes into character values. The **KbdGetCp** function copies the identifier to the variable pointed to by the *pidCodePage* parameter.

Parameters

ulReserved Specifies a reserved value; must be zero.

pidCodePage Points to the variable that receives the code-page identifier. The following are the valid code-page numbers:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The code-page identifier may be any value specified in a **codepage** command in the *config.sys* file. The identifier is 0x0000 if MS OS/2 is using the default translation table for the logical keyboard.

For a description of the possible code-page identifiers and translation tables, see Appendix C, "Country and Code-Page Information."

Example

This example calls the **KbdGetCp** function to identify which code page is being used to translate scan codes for the specified logical keyboard.

```
USHORT idCodePage;
KbdGetCp(0L,                /* must be zero                    */
    &idCodePage,            /* pointer to code-page identifier */
    0);                     /* keyboard handle                 */
```

See Also

DosGetCp, KbdOpen, KbdSetCp

■ KbdGetFocus

USHORT KbdGetFocus(*fWait***,** *hkbd***)**
USHORT *fWait*; /* wait/no-wait flag */
HKBD *hkbd*; /* keyboard handle */

The **KbdGetFocus** function retrieves the focus for the specified logical keyboard. The focus determines which logical keyboard receives keystrokes from the physical keyboard. A logical keyboard cannot receive keystrokes unless it has the focus.

A process can retrieve the focus at any time, but it must wait if the focus is already being used by another process or thread. If a process has the focus, another process cannot receive the focus until the original process frees it by using the **KbdFreeFocus** function. If more than one process is waiting for the focus, MS OS/2 chooses which one receives the focus.

An application must set the focus to an opened keyboard handle before calling functions such as **KbdCharIn**.

Parameters

fWait Specifies whether to wait for the focus to become available. If this parameter is IO_WAIT, the function waits for the focus. If the parameter is IO_NOWAIT, the function returns immediately whether or not it retrieved the focus.

hkbd Identifies the logical keyboard that receives the focus. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_KBD_FOCUS_ALREADY_ACTIVE
ERROR_KBD_UNABLE_TO_FOCUS

Example

This example opens a logical keyboard and calls **KbdGetFocus** to retrieve the focus for the opened keyboard. Once the **KbdFreeFocus** function is called, the focus goes to any process that is waiting for it by calling **KbdGetFocus**. If no process is waiting, MS OS/2 uses the default keyboard:

```
HKBD hkbd;
KbdOpen(&hkbd);
KbdGetFocus(IO_WAIT, hkbd);  /* retrieves focus of logical keyboard */
    .
    .
    .
KbdFreeFocus(hkbd);          /* frees the focus                      */
```

See Also

KbdCharIn, KbdFreeFocus, KbdOpen

■ KbdGetStatus

USHORT KbdGetStatus(*pkbstKbdInfo***,** *hkbd***)**
PKBDINFO *pkbstKbdInfo***;** /* pointer to structure for keyboard status */
HKBD *hkbd***;** /* keyboard handle */

The **KbdGetStatus** function retrieves the status of the specified logical keyboard. The keyboard status specifies the state of the keyboard echo mode, input mode, turnaround character, interim character flags, and shift state.

The **KbdGetStatus** function is a family API function.

Parameters

pkbstKbdInfo Points to the **KBDINFO** structure that receives the keyboard status. The **KBDINFO** structure has the following form:

```
typedef struct _KBDINFO {
    USHORT cb;
    USHORT fsMask;
    USHORT chTurnAround;
    USHORT fsInterim;
    USHORT fsState;
} KBDINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_KBD_INVALID_LENGTH

Comments

Although the initial status of a logical keyboard depends on the system, the logical keyboard typically has echo and ASCII modes turned on, and has a single-byte turnaround character whose value corresponds to the ENTER key.

Restrictions

In real mode, the following restriction applies to the **KbdGetStatus** function:

- Interim and turnaround characters are not supported.

Example

This example calls the **KbdGetStatus** function to retrieve the status of the default keyboard. It then checks to see if echo mode is turned on:

```
KBDINFO kbstInfo;
kbstInfo.cb = sizeof(kbstInfo);         /* length of status buffer */
KbdGetStatus(&kbstInfo, 0);
if (kbstInfo.fsMask & KEYBOARD_ECHO_ON) {
    VioWrtTTY("Echo is on\n\r", 12, 0);
```

See Also

KbdSetStatus, KbdOpen

■ KbdOpen

USHORT KbdOpen(*phkbd***)**
PHKBD *phkbd*; /* pointer to variable for keyboard handle */

The **KbdOpen** function opens a logical keyboard and creates a unique handle that identifies a logical keyboard for use in subsequent **Kbd** (or other MS OS/2) functions. The **KbdOpen** function initializes the logical keyboard to use the default system code page.

Parameters

phkbd Points to the variable that receives the handle of the logical keyboard.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

Any MS OS/2 function that can receive input through a handle (for example, the **DosRead** function) can use the handle created by the **KbdOpen** function.

Example

This example calls the **KbdOpen** function to create and open a handle for a logical keyboard. Before you can access this logical keyboard, you must call the **KbdGetFocus** function to retrieve the focus:

```
HKBD hkbd;
KbdOpen(&hkbd);
KbdGetFocus(IO_WAIT, hkbd);
```

See Also

DosRead, KbdClose, KbdGetFocus

■ KbdPeek

USHORT KbdPeek(*pkbciKeyInfo, hkbd***)**
PKBDKEYINFO *pkbciKeyInfo*; /* pointer to structure for keystroke info. */
HKBD *hkbd*; /* keyboard handle */

The **KbdPeek** function retrieves character and scan-code information from a logical keyboard. The function copies information to the structure pointed to by the *pkbciKeyInfo* parameter. The keystroke information includes the character value of the key, the scan code, the keystroke status, the state of the shift keys, and the system time (in milliseconds) when the keystroke occurred. For information on scan codes, key codes, and MS OS/2 control and editing keys, see Chapter 5, "File Formats."

The **KbdPeek** function is a family API function.

Parameters

pkbciKeyInfo Points to the **KBDKEYINFO** structure that receives the keystroke information. The **KBDKEYINFO** structure has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR chChar;
    UCHAR chScan;
    UCHAR fbStatus;
    UCHAR bNlsShift;
    USHORT fsState;
    ULONG time;
} KBDKEYINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **KbdPeek** function copies but does not remove keystroke information from the input buffer of the specified logical keyboard. Although echo mode for the logical keyboard may be turned on, the **KbdPeek** function does not echo the characters it reads. If the keyboard is in ASCII mode, **KbdPeek** retrieves keystroke information for each key pressed, except shift keys and MS OS/2 CTRL keys. If the keyboard is in binary mode, **KbdPeek** retrieves keystroke information for any key pressed, except shift keys. In most cases, a shift key is pressed in combination with other keys to create a single keystroke. In binary mode with shift report turned on, a shift key by itself creates a keystroke that this function can retrieve. For more information on binary mode and shift-report mode, see the **KbdSetStatus** function.

The **KbdPeek** function retrieves extended ASCII codes, such as when the ALT key and another key, called the primary key, are pressed simultaneously. When the **KbdPeek** function retrieves an extended ASCII code, it sets the **chChar** field of the **KBDKEYINFO** structure to 0x0000 or 0x00E0 and copies the code to the **chScan** field. The extended code is usually the scan code of the primary key. In ASCII mode, the function retrieves only complete extended codes, which means that if both bytes of the extended code do not fit in the buffer, neither byte is retrieved. For more information on extended ASCII codes, see Appendix C, "Country and Code-Page Information."

The **KbdPeek** function must be called twice to retrieve a code for a double-byte character set (DBCS). If the code retrieved is the first byte of a double-byte character, the **fbStatus** field of the **KBDKEYINFO** structure is set to 0x0080.

Restrictions

In real mode, the following restrictions apply to the **KbdPeek** function:

- It does not copy the system time to the **KBDKEYINFO** structure, and there is no interim character support.
- It retrieves characters only from the default logical keyboard (handle 0).
- The **fbStatus** field may be 0x0000 or SHIFT_KEY_IN.
- The *hkbd* parameter is ignored.

Example

This example calls the **KbdPeek** function to read a character from the default keyboard without removing it from the keyboard input buffer. If there is already a character in the buffer, the **fbStatus** field specifies this by setting the sixth bit (0x40):

```
KBDKEYINFO kbciKeyInfo;
    .
    .
    .
KbdPeek(&kbciKeyInfo, 0);
if (kbciKeyInfo.fbStatus & 0x40) {
```

See Also

KbdCharIn, KbdGetStatus, KbdOpen, KbdSetStatus

■ KbdRegister

```
USHORT KbdRegister(pszModuleName, pszEntryName, fFunctions)
PSZ pszModuleName;    /* pointer to string for module name      */
PSZ pszEntryName;     /* pointer to string for entry-point name */
ULONG fFunctions;     /* function flags                         */
```

The **KbdRegister** function registers a **Kbd** subsystem for the specified logical keyboard. The function temporarily replaces the one or more default **Kbd** functions, as specified by the *fFunctions* parameter, with the function(s) in the module named by the *pszModuleName* parameter. Once **KbdRegister** replaces a function, MS OS/2 passes any subsequent call to the replaced function to a function in the given module. If you do not replace a function, MS OS/2 continues to call the default **Kbd** function.

Parameters

pszModuleName Points to the null-terminated string that contains the name of the dynamic-link module specifying the replacement **Kbd** functions. The string must be a valid filename.

pszEntryName Points to the null-terminated string that contains the dynamic-link entry-point name of the function that replaces the specified **Kbd** function(s). For a full description, see the following "Comments" section.

fFunctions Specifies the flags for the function(s) to replace. This parameter can be any combination of the following values:

Value	Meaning
KR_KBDCHARIN	Replace **KbdCharIn**.
KR_KBDPEEK	Replace **KbdPeek**.
KR_KBDFLUSHBUFFER	Replace **KbdFlushBuffer**.
KR_KBDGETSTATUS	Replace **KbdGetStatus**.
KR_KBDSETSTATUS	Replace **KbdSetStatus**.
KR_KBDSTRINGIN	Replace **KbdStringIn**.
KR_KBDOPEN	Replace **KbdOpen**.
KR_KBDCLOSE	Replace **KbdClose**.
KR_KBDGETFOCUS	Replace **KbdGetFocus**.
KR_KBDFREEFOCUS	Replace **KbdFreeFocus**.
KR_KBDGETCP	Replace **KbdGetCp**.

Value	Meaning
KR_KBDSETCP	Replace **KbdSetCp**.
KR_KBDXLATE	Replace **KbdXlate**.
KR_KBDSETCUSTXT	Replace **KbdSetCustXt**.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_KBD_INVALID_ASCIIZ
ERROR_KBD_INVALID_MASK
ERROR_KBD_REGISTER

Comments

MS OS/2 passes a **Kbd** function to the given module by preparing the stack and calling the function pointed to by the *pszEntryName* parameter. The specified module must export the entry-point function name. The entry-point function must check the function code on the stack to determine which function is being requested, and then pass control to the appropriate function in the module. The entry-point function may then access any additional parameters placed on the stack by the original call to **KbdRegister**.

Only one process in a screen group may use the **KbdRegister** function at any given time. That is, only one process can replace **Kbd** functions at any given time. The process can restore the default **Kbd** functions by calling the **KbdDeRegister** function. A process can replace **Kbd** functions any number of times, but it may do so only by first restoring the default functions, and then reregistering the new functions.

The entry-point function (*FuncName*) must have the following form:

SHORT FAR *FuncName*(*selDataSeg*, *usReserved1*, *fFunction*,
ulReserved2, *usParam1*, *usParam2*, *usParam3*, *usParam4*,
usParam5, *usParam6*)
SEL *selDataSeg*;
USHORT *usReserved1*;
USHORT *fFunction*;
ULONG *ulReserved2*;
USHORT *usParam1*;
USHORT *usParam2*;
USHORT *usParam3*;
USHORT *usParam4*;
USHORT *usParam5*;
USHORT *usParam6*;

Parameters	Description
selDataSeg	Specifies the data-segment selector of the process that calls the **Kbd** function.
usReserved1	Specifies a reserved value that must not be changed. This value represents a return address for the MS OS/2 function that routes **Kbd** function calls.

Parameters	Description
fFunction	Specifies the function code of the function request. This parameter can be one of the following values:

Value	Meaning
0x0000	**KbdCharIn** called.
0x0001	**KbdPeek** called.
0x0002	**KbdFlushBuffer** called.
0x0003	**KbdGetStatus** called.
0x0004	**KbdSetStatus** called.
0x0005	**KbdStringIn** called.
0x0006	**KbdOpen** called.
0x0007	**KbdClose** called.
0x0008	**KbdGetFocus** called.
0x0009	**KbdFreeFocus** called.
0x000A	**KbdGetCp** called.
0x000B	**KbdSetCp** called.
0x000C	**KbdXlate** called.
0x000D	**KbdSetCustXt** called.

Parameters	Description
ulReserved2	Specifies a reserved value that must not be changed. This parameter represents the return address of the program that calls the specified **Kbd** function.
usParam1-usParam6	Specify up to six unsigned values passed with the call to the **Kbd** function. The number and type of parameters used depend on the specific function.

The entry-point function should determine which function is requested and then carry out an appropriate action by using the passed parameters. If necessary, the entry-point function can call a function within the same module to carry out the task. The entry-point or replacement function must leave the stack in the same state as it was received. This is required since the return addresses on the stack must be available in the correct order to return control to the program that originally called the **KbdRegister** function.

The registered function should return -1 if it wants the original function called, 0 if no error occurred, or an error value.

In general, if the replacement function needs to access the keyboard, it must use the input-and-output control functions for the keyboard. For more information, see Chapter 3, "Input-and-Output Control Functions."

The **KbdRegister** function itself cannot be replaced.

See Also

KbdDeRegister, KbdFlushBuffer

KbdSetCp

USHORT KbdSetCp(*usReserved, idCodePage, hkbd***)**
USHORT *usReserved*; /* must be zero */
USHORT *idCodePage*; /* code-page identifier */
HKBD *hkbd*; /* keyboard handle */

The **KbdSetCp** function sets the code-page identifier for the specified logical keyboard. The code-page identifier defines which translation table MS OS/2 uses to translate keystrokes into character values. The code-page identifier may be any value specified in a **codepage** command in the *config.sys* file, or 0x0000 for the default translation table for the logical keyboard.

The **KbdSetCp** function also clears the input buffer of the logical keyboard.

Parameters

usReserved Specifies a reserved value; must be zero.

idCodePage Specifies the code-page identifier. If the identifier is 0x0000, the default translation table is used. The following are the valid code-page numbers:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

For a description of the possible code-page identifiers and translation tables, see Appendix C, "Country and Code-Page Information."

Example

This example calls **KbdSetCp** to change the **Kbd** subsystem so that it uses the U.S. multilingual code page (850) when translating keystrokes for the default keyboard. The code page must be installed by the *config.sys* file or this function returns an error value:

```
KbdSetCp(0,                     /* reserved             */
    850,                        /* code-page identifier */
    0);                         /* keyboard handle      */
```

See Also

DosSetCp, KbdGetCp, KbdOpen, KbdSetCustXt

KbdSetCustXt

```
USHORT KbdSetCustXt(pusTransTbl, hkbd)
PUSHORT pusTransTbl;      /* pointer to translation table */
HKBD hkbd;                /* keyboard handle              */
```

The **KbdSetCustXt** function installs a custom translation table for the specified logical keyboard. MS OS/2 uses the translation table to generate character values for all subsequent keystrokes from the logical keyboard.

The **KbdSetCustXt** function does not copy the translation table, so the process must maintain the table in memory while it is in use, where it remains until the process calls the **KbdSetCp** or **KbdSetCustXt** function to set another translation table, or calls the **KbdClose** function to close the logical keyboard.

Parameters

pusTransTbl Points to the translation table. The table has the size and format described in Appendix C, "Country and Code-Page Information."

hkbd Identifies the logical keyboard that uses the new code page. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

DosSetCp, KbdClose, KbdOpen, KbdSetCp, KbdXlate

KbdSetFgnd

```
USHORT KbdSetFgnd(void)
```

The **KbdSetFgnd** function raises the priority of the foreground keyboard's thread. This function is used by a **Kbd** subsystem, not by an application.

Parameters

This function has no parameters.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

KbdSetStatus

```
USHORT KbdSetStatus(pkbstKbdInfo, hkbd)
PKBDINFO pkbstKbdInfo;    /* pointer to structure for keyboard status */
HKBD hkbd;                /* keyboard handle                          */
```

The **KbdSetStatus** function sets the status for the specified logical keyboard. The keyboard status specifies the state of the keyboard echo mode, input mode, turnaround character, interim character flags, and shift state.

The **KbdSetStatus** function is a family API function.

Parameters

pkbstKbdInfo Points to the **KBDINFO** structure that contains the keyboard status. The **KBDINFO** structure has the following form:

```
typedef struct _KBDINFO  {
    USHORT cb;
    USHORT fsMask;
    USHORT chTurnAround;
    USHORT fsInterim;
    USHORT fsState;
} KBDINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_KBD_INVALID_ECHO_MASK
ERROR_KBD_INVALID_INPUT_MASK
ERROR_KBD_INVALID_LENGTH

Comments

In most cases, a shift key is pressed in combination with other keys to create a single keystroke. In binary mode with shift report turned on, a shift key by itself creates a keystroke that the **KbdCharIn** or **KbdPeek** function can retrieve.

Restrictions

In real mode, the following restrictions apply to the **KbdSetStatus** function:

- Interim and turnaround characters are not supported.
- Binary mode with echo mode on is not supported.
- The *hkbd* parameter is ignored.

Example

This example retrieves the current status of the default keyboard, masks the ASCII-mode bit, uses the OR operator to set the binary-mode bit, and calls the **KbdSetStatus** function to change the keyboard status to binary mode:

```
KBDINFO kbstInfo;
kbstInfo.cb = sizeof(kbstInfo);
KbdGetStatus(&kbstInfo, 0);          /* gets current status   */
kbstInfo.fsMask =
    (kbstInfo.fsMask & 0x00F7)       /* masks out ASCII mode */
    | 0x0004;                        /* OR into binary mode  */
KbdSetStatus(&kbstInfo, 0);          /* sets new status      */
```

See Also

KbdCharIn, KbdGetStatus, KbdOpen, KbdPeek

KbdStringIn

```
USHORT KbdStringIn(pchBuffer, psibLength, fWait, hkbd)
PCH pchBuffer;                 /* pointer to buffer for string          */
PSTRINGINBUF psibLength;       /* pointer to structure for string length */
USHORT fWait;                  /* wait/no-wait flag                     */
HKBD hkbd;                     /* keyboard handle                       */
```

The **KbdStringIn** function reads a string of characters from a logical keyboard. The function copies the character value of each keystroke to the buffer pointed to by the *pchBuffer* parameter. Depending on the input mode of the keyboard and on the value of the *fWait* parameter, **KbdStringIn** continues to copy characters until it fills the buffer, retrieves the turnaround character, or reaches the end of the buffer.

The **KbdStringIn** function is a family API function.

Parameters

pchBuffer Points to the buffer that receives the character string.

psibLength Points to the **STRINGINBUF** structure that contains the length of the buffer that receives the string. The **STRINGINBUF** structure has the following form:

```
typedef struct _STRINGINBUF {
    USHORT cb;
    USHORT cchIn;
} STRINGINBUF;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

fWait Specifies whether to wait for the entire string to be read. If this parameter is IO_WAIT, the function waits for all characters up to the next turnaround character or until it reaches the end of the buffer. If the parameter is IO_NOWAIT, the function returns immediately with whatever characters are available.

hkbd Identifies the logical keyboard to read from. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **KbdStringIn** function removes keystroke information from the input buffer of the specified logical keyboard as it copies a character. If echo and ASCII modes are turned on, the function echoes characters on the screen as they are typed. If the keyboard is in ASCII mode, the function retrieves a character for each key pressed, except shift keys and MS OS/2 CTRL and editing keys. If the keyboard is in binary mode, the function retrieves a character for any key pressed except shift keys.

The **KbdStringIn** function can retrieve extended ASCII codes, such as when the ALT key, and another key, called the primary key, are pressed simultaneously. When the function retrieves an extended code, the first character is 0x0000 or 0x00E0 and the second is the extended code. The extended code is usually the scan code of the primary key. In ASCII mode, the function retrieves only complete extended codes, which means that if both bytes of the extended code do not fit in the buffer, neither byte is retrieved. For more information on extended ASCII codes, see Appendix C, "Country and Code-Page Information."

In ASCII mode, **KbdStringIn** recognizes the MS OS/2 editing keys. These keys can be used to display and edit the previously entered string. The **KbdStringIn** function permits editing of the previous string only if the **cchIn** field of the **STRINGINBUF** structure is set to the length of the previous string before the function is called. If this field is set to zero, the line cannot be edited.

Restrictions

In real mode, the following restriction applies to the **KbdStringIn** function:

- The *hkbd* parameter is ignored.

Example

This example calls the **KbdStringIn** function to read a character string from the default keyboard. In ASCII mode, the function waits for the RETURN key to be pressed; in binary mode, it waits for the buffer to be filled:

```
CHAR achBuf[40];
STRINGINBUF kbsiBuf;
kbsiBuf.cb = sizeof(achBuf);
KbdStringIn(achBuf,              /* address of buffer           */
    &kbsiBuf,                    /* address of length structure */
    IO_WAIT,                     /* waits for characters        */
    0);                          /* keyboard handle             */
VioWrtTTY("\n", 1, 0);           /* sends linefeed character    */
VioWrtTTY(achBuf, kbsiBuf.cchIn, 0); /* displays string         */
```

See Also

DosRead, KbdCharIn, KbdGetStatus, KbdOpen, KbdSetStatus

■ KbdSynch

USHORT KbdSynch(*fWait***)**
USHORT *fWait*; /* wait/no-wait flag */

The **KbdSynch** function synchronizes access to the keyboard device driver.

This function should be used by a **Kbd** subsystem, not by an application. You cannot replace the **KbdSynch** function by using the **KbdRegister** function.

Parameters

fWait Specifies whether to wait for access to the keyboard router if access is not available. If this parameter is IO_WAIT, the function waits for access to the keyboard router. If the parameter is IO_NOWAIT, the function does not wait and returns immediately.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **KbdSynch** function requests an exclusive system semaphore that blocks all other threads within a screen group until the semaphore is cleared. This semaphore is cleared when a called **Kbd** function returns.

See Also

DosDevIOCtl, KbdRegister

■ KbdXlate

USHORT KbdXlate(*pkbxlKeyStroke, hkbd***)**
PKBDXLATE *pkbxlKeyStroke*; /* pointer to structure for scan code */
HKBD *hkbd*; /* keyboard handle */

The **KbdXlate** function translates a scan code and its shift states into a character value. The function uses the current translation table of the specified logical keyboard.

In order to be translated, accent-key combinations, double-byte characters, and extended ASCII characters may require several calls to the **KbdXlate** function.

Parameters

pkbxlKeyStroke Points to the **KBDTRANS** structure that contains the scan code to translate. It also receives the character value when the function returns. The **KBDTRANS** structure has the following form:

```
typedef struct _KBDTRANS {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
    USHORT fsDD;
    USHORT fsXlate;
    USHORT fsShift;
    USHORT sZero;
} KBDTRANS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hkbd Identifies the logical keyboard. The handle must have been created previously by using the **KbdOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

DosMonReg, KbdOpen, KbdSetCustXt

MouClose

USHORT MouClose(*hmou*)
HMOU *hmou*; /* mouse handle */

The **MouClose** function closes the mouse identified by the given handle. The function removes the mouse pointer from the screen only if the process is the last one in the screen group to have the mouse open.

Parameters

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle then calls the **MouClose** function to close the open handle:

```
HMOU hmou;
MouOpen(OL, &hmou);
     .
     .
     .
MouClose(hmou);
```

See Also

MouOpen

MouDeRegister

USHORT MouDeRegister(*void*)

The **MouDeRegister** function restores the default **Mou** subsystem functions and releases any previously registered **Mou** subsystem. This function restores the default **Mou** subsystem for all processes in the current screen group.

Once a process registers a **Mou** subsystem, no other process in the screen group may register a **Mou** subsystem until the default subsystem is restored. Only the process that registers a **Mou** subsystem may call the **MouDeRegister** function to restore the default subsystem.

Parameters

This function has no parameters.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_DEREGISTER

See Also

MouRegister

MouDrawPtr

USHORT MouDrawPtr(*hmou***)**
HMOU *hmou*; /* mouse handle */

The **MouDrawPtr** function enables the mouse pointer to be drawn on the screen, using the pointer shape defined by the most recent call to the **MouSetPtrShape** function. The **MouDrawPtr** function releases any exclusion rectangle that may have been previously set by using the **MouRemovePtr** function. An exclusion rectangle defines a rectangular region of the screen in which MS OS/2 will not display the pointer.

Parameters

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Comments

The **MouDrawPtr** function does not itself draw the mouse pointer. Instead, it directs MS OS/2 to call the mouse device driver at each mouse interrupt. If the mouse device driver has been disabled (by the **MouSetDevStatus** function), **MouDrawPtr** releases the current exclusion rectangle but does not draw the pointer.

Example

This example creates a mouse handle then calls the **MouDrawPtr** function to enable the mouse pointer to be drawn on the screen:

```
HMOU hmou;
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
```

See Also

MouOpen, MouRemovePtr, MouSetDevStatus, MouSetPtrShape

MouFlushQue

USHORT MouFlushQue(*hmou***)**
HMOU *hmou*; /* mouse handle */

The **MouFlushQue** function removes any existing mouse events from the mouse event queue.

Parameters

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle then calls the **MouFlushQue** function to remove any events from the existing mouse event queue:

```
HMOU hmou;
MouOpen(OL, &hmou);
    .
    .
    .
MouFlushQue(hmou);
```

See Also

MouGetNumQueEl, MouOpen, MouReadEventQue

MouGetDevStatus

```
USHORT MouGetDevStatus(pfsDevStatus, hmou)
PUSHORT pfsDevStatus;    /* pointer to buffer for status */
HMOU hmou;               /* mouse handle                 */
```

The **MouGetDevStatus** function retrieves the device status for the specified mouse.

Parameters

pfsDevStatus Points to the variable that receives the device status. It can be any combination of the following values:

Value	Meaning
MOUSE_QUEUEBUSY	Event queue is busy with input/output (I/O).
MOUSE_BLOCKREAD	Block read is in progress.
MOUSE_FLUSH	Flush buffer is in progress.
MOUSE_UNSUPPORTED_MODE	Mouse device driver is disabled because of unsupported mode.
MOUSE_DISABLED	Mouse device driver is disabled.
MOUSE_MICKEYS	Mouse motion is given in mickeys, not in pels.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle then calls the **MouGetDevStatus** function to retrieve the status for the mouse identified by the handle:

```
USHORT fsDevStatus;
HMOU hmou;
MouOpen(OL, &hmou);
MouGetDevStatus(&fsDevStatus, hmou);
if (fsDevStatus & MOUSE_DISABLED ||
        fsDevStatus & MOUSE_UNSUPPORTED_MODE)
    VioWrtTTY("mouse is disabled\r\n", 19, 0);
```

See Also

MouOpen, MouSetDevStatus

MouGetEventMask

USHORT MouGetEventMask(*pfsEvents***,** *hmou***)**
PUSHORT *pfsEvents*; /* pointer to buffer for event mask */
HMOU *hmou*; /* mouse handle */

The **MouGetEventMask** function retrieves the event mask for the specified mouse. The event mask specifies the user actions that cause MS OS/2 to generate mouse events. MS OS/2 responds to a user action by copying a mouse event to the event queue.

Parameters

pfsEvents Points to the variable that receives the event mask. It can be any combination of the following values:

Value	Meaning
MOUSE_MOTION	Mouse motion.
MOUSE_MOTION_WITH_BN1_DOWN	Mouse motion with button-1-down event.
MOUSE_BN1_DOWN	Button-1-down event.
MOUSE_MOTION_WITH_BN2_DOWN	Mouse motion with button-2-down event.
MOUSE_BN2_DOWN	Button-2-down event.
MOUSE_MOTION_WITH_BN3_DOWN	Mouse motion with button-3-down event.
MOUSE_BN3_DOWN	Button-3-down event.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Comments

Button 1 is the left button on the mouse.

Example

This example creates a mouse handle, calls the **MouGetEventMask** function, and checks the event mask to see if events are accepted from a third button on the mouse:

```
HMOU hmou;
USHORT fsEvents;
MouOpen(0L, &hmou);
MouGetEventMask(&fsEvents, hmou);
if(fsEvents & (MOUSE_MOTION_WITH_BN3_DOWN | MOUSE_BN3_DOWN)
    VioWrtTTY("Three buttons enabled\n\r", 23, 0);
```

See Also

MouOpen, MouReadEventQue, MouSetEventMask

MouGetNumButtons

```
USHORT MouGetNumButtons(pusButtons, hmou)
PUSHORT pusButtons;     /* pointer to variable for number of mouse buttons */
HMOU hmou;              /* mouse handle                                     */
```

The **MouGetNumButtons** function retrieves the number of buttons on the current mouse.

Parameters

pusButtons Points to the variable that receives the number of buttons on the mouse.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle then calls the **MouGetNumButtons** function to retrieve the number of mouse buttons:

```
HMOU hmou;
USHORT usButtons;
MouOpen(0L, &hmou);
MouGetNumButtons(&usButtons, hmou);
if(usButtons == 2)
    VioWrtTTY("Your mouse has two buttons\n\r", 28, 0);
```

See Also

MouOpen

MouGetNumMickeys

```
USHORT MouGetNumMickeys(pusMickeys, hmou)
PUSHORT pusMickeys;     /* pointer to variable for mickeys per centimeter */
HMOU hmou;              /* mouse handle                                    */
```

The **MouGetNumMickeys** function retrieves the number of mickeys that the specified mouse travels for each centimeter of motion. A mickey is the smallest unit of motion a mouse can measure. The number of mickeys per centimeter for a mouse depends on the device and may also depend on the current setting of the device.

Parameters

pusMickeys Points to the variable that receives the number of mickeys per centimeter.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle then calls the **MouGetNumMickeys** function to retrieve the current number of mickeys per centimeter:

```
HMOU hmou;
USHORT usMickeys;
MouOpen (OL, &hmou);
MouGetNumMickeys(&usMickeys, hmou);
```

See Also

MouOpen

■ MouGetNumQueEl

USHORT MouGetNumQueEl(*pmouqi, hmou***)**
PMOUQUEINFO *pmouqi*; /* pointer to structure for number of events */
HMOU *hmou*; /* mouse handle */

The **MouGetNumQueEl** function retrieves the number of events in the mouse event queue.

Parameters

pmouqi Points to the **MOUQUEINFO** structure that receives the number of events in the mouse event queue. The **MOUQUEINFO** structure has the following form:

```
typedef struct _MOUQUEINFO {
    USHORT cEvents;
    USHORT cmaxEvents;
} MOUQUEINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle, enables the mouse pointer to be drawn, and runs within an infinite **for** loop until there are no events in the queue:

```
HMOU hmou;
MOUEVENTINFO mouevEvent;
MOUQUEINFO mouqi;
USHORT fWait = FALSE;
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
for (;;) {
    MouGetNumQueEl(&mouqi,              /* retrieves queue        */
        hmou;
    if (mouqi.cEvents > 1)              /* until the last queue... */
        MouReadEventQue(&mouevEvent, &fWait, hmou);
    else
        break;
}
```

See Also

MouFlushQue, MouOpen, MouReadEventQue

■ MouGetPtrPos

USHORT MouGetPtrPos(*pmouplPosition*, *hmou*)
PPTRLOC *pmoupl*; /* pointer to structure for current mouse position */
HMOU *hmou*; /* mouse handle */

The **MouGetPtrPos** function retrieves the current position of the mouse device. This position is given in screen coordinates.

Parameters

pmoupl Points to the **PTRLOC** structure that receives the coordinates of the mouse position. The **PTRLOC** structure has the following form:

```
typedef struct _PTRLOC {
    USHORT row;
    USHORT col;
} PTRLOC;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Comments

The current device status as defined by the **MouSetDevStatus** function does not affect the **row** and **col** fields of the **PTRLOC** structure. These fields always specify an absolute position relative to the upper-left corner of the screen.

Example

This example creates a mouse handle and enables the mouse pointer to be drawn. It then displays the text "Place mouse here" at the top of the screen and repeatedly calls the **MouGetPtrPos** function until the mouse is moved over the text:

```
PTRLOC moupl;
HMOU hmou;
BYTE bAttr = 0x72;        /* green character on white background */
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
VioWrtCharStrAtt("Place mouse here", 16, 0, 35, &bAttr, 0);
do
    MouGetPtrPos(&moupl, hmou);
while (moupl.row != 0 || (moupl.col < 35 || moupl.col > 50));
```

See Also

MouOpen, MouSetDevStatus, MouSetPtrPos

■ MouGetPtrShape

USHORT MouGetPtrShape(*pbBuffer*, *pmoupsInfo*, *hmou*)
PBYTE *pbBuffer*; /* pointer to buffer for shape masks */
PPTRSHAPE *pmoupsInfo*; /* pointer to structure for shape information */
HMOU *hmou*; /* mouse handle */

The **MouGetPtrShape** function retrieves the AND and XOR masks that define the shape of the pointer for the specified mouse. **MouGetPtrShape** also retrieves information about the pointer, such as the width and height of masks and the location of the hot spot.

Parameters

pbBuffer Points to the buffer that receives the masks.

pmoupsInfo Points to the **PTRSHAPE** structure that receives the pointer information. The **PTRSHAPE** structure has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE

Comments

The **MouGetPtrShape** function copies the AND and XOR masks to the buffer pointed to by the *pbBuffer* parameter. The format and size of the masks depend on the display device and the video mode. In text mode, each mask is typically a character/attribute pair. In graphics mode, each mask is a bitmap.

The **MouGetPtrShape** function copies information about the pointer to the structure pointed to by the *pmoupsInfo* parameter. This structure defines the length (in bytes) of the AND and XOR masks, the width and height of each mask, and the offset from the current mouse position (or hot spot) to the upper-left corner of the pointer shape.

Before calling **MouGetPtrShape**, you must set the **cb** field of the **PTRSHAPE** structure to the appropriate buffer size. If the field does not specify an appropriate size, the function copies the current size to the field and returns an error without copying the masks to the specified buffer.

Example

This example creates a mouse handle, draws the mouse pointer, and calls the **MouGetPtrShape** function to retrieve the shape of the mouse pointer:

```
PTRSHAPE moupsInfo;
BYTE abBuffer[4];
HMOU hmou;
MouOpen(0L, &hmou);
MouDrawPtr(hmou);
moupsInfo.cb = sizeof(abBuffer);
MouGetPtrShape(abBuffer, &moupsInfo, hmou);
```

See Also

MouOpen, MouSetPtrShape

■ MouGetScaleFact

USHORT MouGetScaleFact(*pmouscFactors, hmou***)**
PSCALEFACT *pmouscFactors*; /* pointer to structure for scaling factors */
HMOU *hmou*; /* mouse handle */

The **MouGetScaleFact** function retrieves the horizontal and vertical scaling factors for the specified mouse. The scaling factors define the number of mickeys the mouse must travel horizontally or vertically in order to cause MS OS/2 to move the mouse pointer one screen unit.

Parameters

pmouscFactors Points to the **SCALEFACT** structure that receives the scaling factors. The **SCALEFACT** structure has the following form:

```
typedef struct _SCALEFACT {
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle then calls the **MouGetScaleFact** function to retrieve the scaling factors for the row and column coordinates:

```
SCALEFACT mouscFactors;
HMOU hmou;
MouOpen(OL, &hmou);
MouGetScaleFact(&mouscFactors, hmou);  /* retrieves scaling factors */
```

See Also

MouGetNumMickeys, MouOpen, MouSetScaleFact

■ MouInitReal

USHORT MouInitReal(*pszDriverName***)**
PSZ *pszDriverName*; /* pointer to string for name of mouse device driver */

The **MouInitReal** function loads and initializes the real-mode mouse device driver pointed to by the *pszDriverName* parameter. You must specify the name of the mouse device driver by using a **device** command in the *config.sys* file.

This function is used only by the Task Manager.

Parameters

pszDriverName Points to the null-terminated string that specifies the name of the mouse device driver. The name must be a valid filename. You can initialize the default mouse device driver by setting this parameter to zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Comments The **Mou** functions are not available in real-mode programs. Instead, all real-mode mouse input and output must be carried out using the real-mode (**int 33h**) interface.

See Also **MouOpen**

MouOpen

```
USHORT MouOpen(pszDriverName, phmou)
PSZ pszDriverName;    /* pointer to mouse driver name          */
PHMOU phmou;          /* pointer to variable for mouse handle */
```

The **MouOpen** function opens the mouse for the current screen group and creates a handle that can be used in subsequent **Mou** functions (to display the mouse pointer, retrieve the current location of the mouse pointer, etc.).

The **MouOpen** function creates the mouse handle for the current screen group only. Any number of processes may open this handle, but all processes in the screen group share it. For example, if one process changes the color of the mouse pointer, the pointer color changes for all other processes in the same screen group.

When the mouse handle is first created, **MouOpen** does not display the mouse pointer. The **MouDrawPtr** function must be called to display the pointer. (A mouse device driver is required to draw the pointer. If the mouse device driver pointed by the *pszDriverName* parameter does not exist or cannot be opened, an error occurs and the pointer is not drawn. If *pszDriverName* is set to zero, the default mouse device driver is used; that is, the driver specified in a **device** command in the *config.sys* file is used.)

Parameters *pszDriverName* Points to the null-terminated string that contains the name of the mouse device driver. The name must be a valid filename. If this parameter is set to zero, the default pointer-draw driver is used.

phmou Points to the variable that receives the mouse handle.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_MODULE
ERROR_MOUSE_NO_DEVICE

Example This example calls the **MouOpen** function to create a mouse handle to be used by the current screen group:

```
HMOU hmou;
MouOpen(0L, &hmou);
```

See Also **MouClose, MouDrawPtr**

■ MouReadEventQue

```
USHORT MouReadEventQue(pmouevEvent, pfWait, hmou)
PMOUEVENTINFO pmouevEvent;    /* pointer to structure for mouse event */
PUSHORT pfWait;               /* wait/no-wait flag                    */
HMOU hmou;                    /* mouse handle                         */
```

The **MouReadEventQue** function retrieves a mouse event from the event queue of the specified mouse. The event queue is a buffer to which MS OS/2 copies each mouse event. A mouse event is a structure that specifies the user action that generated the event, the location of the mouse when the event occurred, and system time when the event occurred.

MS OS/2 copies a mouse event to the event queue whenever the user moves the mouse or presses or releases a mouse button. The mouse event can specify a single action or a combination of actions, such as the mouse being moved with a button down. MS OS/2 copies a mouse event for a given action only if the event mask enables reporting for that action. For more information, see the **MouSetEventMask** function.

Parameters

pmouevEvent Points to the **MOUEVENTINFO** structure that receives the mouse event. The **MOUEVENTINFO** structure has the following form:

```
typedef struct _MOUEVENTINFO {
    USHORT fs;
    ULONG time;
    USHORT row;
    USHORT col;
} MOUEVENTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pfWait Points to the variable that specifies whether the function waits for an event. If this parameter is MOU_NOWAIT and the queue is empty, the function fills the **MOUEVENTINFO** structure with zeros and returns immediately. If the parameter is MOU_WAIT, the function waits for a mouse event if none is available.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE
NO_ERROR_MOUSE_NO_DATA

Comments

Button 1 is the left button on the mouse.

The meaning of the **row** and **col** fields of the **MOUEVENTINFO** structure depends on the current device status as defined by the most recently used **MouSetDevStatus** function. The values may be absolute or relative, and the units may be mickeys, character cells, or pels.

Although a specific action may not generate a mouse event, the **fs** field of the **MOUEVENTINFO** structure may include information about the action when some other event occurs. For example, even if button 2 is disabled, **fs** is set to 0x0014 if the user presses button 1 when button 2 is also down. If the *pfWait* parameter is MOU_NOWAIT, **fs** will be zero if either a mouse-button-up event occurs or no event occurs. To see whether an event occurred, check the **time** field; it will be zero if there was no event.

Example

This example creates a mouse handle, enables the mouse pointer to be drawn, and calls the **MouReadEventQue** function, telling it to wait until a mouse event occurs. If the mouse event is the left mouse button down, the message "Left Button" is displayed:

```
MOUEVENTINFO mouevEvent;
HMOU hmou;
USHORT fWait = TRUE;                    /* waits for mouse event      */
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
MouReadEventQue(&mouevEvent, &fWait, hmou);
if (mouevEvent.fs & 0x04)               /* if left button pressed... */
    VioWrtTTY("Left Button\n\r", 13, 0);
```

See Also

MouGetNumQueEl, MouOpen, MouSetDevStatus, MouSetEventMask

■ MouRegister

USHORT MouRegister(*pszModuleName***,** *pszEntryName***,** *flFunctions***)**

PSZ *pszModuleName*;	/* pointer to string for module name	*/
PSZ *pszEntryName*;	/* pointer to string for entry name	*/
ULONG *flFunctions*;	/* function flags	*/

The **MouRegister** function registers a **Mou** subsystem for the specified mouse. The function temporarily replaces the one (or more) default **Mou** functions, as specified by the *flFunctions* parameter, with the functions in the module pointed to by the *pszModuleName* parameter. Once **MouRegister** replaces a function, MS OS/2 passes any subsequent calls to the replaced function to a function in the given module. If you do not replace a function, MS OS/2 continues to call the default **Mou** function.

Parameters

pszModuleName Points to the null-terminated string that contains the name of the dynamic-link module containing the replacement **Mou** functions.

pszEntryName Points to the null-terminated string that contains the dynamic-link entry-point name of the function that replaces the specified **Mou** function. For a full description, see the following "Comments" section.

flFunctions Specifies the flags of the **Mou** functions to replace. It can be any combination of the following values:

Value	Meaning
MR_MOUGETNUMBUTTONS	Replace **MouGetNumButtons**.
MR_MOUGETNUMMICKEYS	Replace **MouGetNumMickeys**.
MR_MOUGETDEVSTATUS	Replace **MouGetDevStatus**.
MR_MOUGETNUMQUEEL	Replace **MouGetNumQueEl**.
MR_MOUREADEVENTQUE	Replace **MouReadEventQue**.

Value	Meaning
MR_MOUGETSCALEFACT	Replace **MouGetScaleFact**.
MR_MOUGETEVENTMASK	Replace **MouGetEventMask**.
MR_MOUSETSCALEFACT	Replace **MouSetScaleFact**.
MR_MOUSETEVENTMASK	Replace **MouSetEventMask**.
MR_MOUOPEN	Replace **MouOpen**.
MR_MOUCLOSE	Replace **MouClose**.
MR_MOUGETPTRSHAPE	Replace **MouGetPtrShape**.
MR_MOUSETPTRSHAPE	Replace **MouSetPtrShape**.
MR_MOUDRAWPTR	Replace **MouDrawPtr**.
MR_MOUREMOVEPTR	Replace **MouRemovePtr**.
MR_MOUGETPTRPOS	Replace **MouGetPtrPos**.
MR_MOUSETPTRPOS	Replace **MouSetPtrPos**.
MR_MOUINITREAL	Replace **MouInitReal**.
MR_MOUSETDEVSTATUS	Replace **MouSetDevStatus**.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INVALID_ASCIIZ
ERROR_MOUSE_INVALID_MASK
ERROR_MOUSE_REGISTER

Comments

MS OS/2 passes a **Mou** function to the given module by preparing the stack and calling the function pointed to by the *pszEntryName* parameter. Specified module must export the entry-point function name. The entry-point function must check the function code on the stack to determine which function is being requested, then pass control to the appropriate function in the module. The entry-point function may then access any additional parameters placed on the stack by the **MouRegister** function.

Only one process in a screen group may use the **MouRegister** function at any given time. That is, only one process at a time can replace **Mou** functions. The process can restore the default **Mou** functions by calling the **MouDeRegister** function. A process can replace a **Mou** function any number of times, but only by first restoring the default functions and then reregistering the new functions.

The entry-point function (*FuncName*) must have the following form:

SHORT FAR *FuncName*(*usReserved1*, *usFunction*, *ulReserved2*,
    usParam1, *usParam2*, *usParam3*, *usParam4*, *usParam5*)
USHORT *usReserved1*;
USHORT *usFunction*;
ULONG *ulReserved2*;
USHORT *usParam1*;
USHORT *usParam2*;
USHORT *usParam3*;
USHORT *usParam4*;
USHORT *usParam5*;

Parameter	Description
usReserved1	Specifies a reserved value that must not be changed. This value represents a return address for the MS OS/2 function that routes **Mou** function calls.
usFunction	Specifies the function code that identifies the function request. It can be one of the following values:

Value	Meaning
0x0000	**MouGetNumButtons** called.
0x0001	**MouGetNumMickeys** called.
0x0002	**MouGetDevStatus** called.
0x0003	**MouGetNumQueEl** called.
0x0004	**MouReadEventQue** called.
0x0005	**MouGetScaleFact** called.
0x0006	**MouGetEventMask** called.
0x0007	**MouSetScaleFact** called.
0x0008	**MouSetEventMask** called.
0x0009	**MouGetHotKey** called.
0x000A	**MouSetHotKey** called.
0x000B	**MouOpen** called.
0x000C	**MouClose** called.
0x000D	**MouGetPtrShape** called.
0x000E	**MouSetPtrShape** called.
0x000F	**MouDrawPtr** called.
0x0010	**MouRemovePtr** called.
0x0011	**MouGetPtrPos** called.
0x0012	**MouSetPtrPos** called.
0x0013	**MouInitReal** called.
0x0014	**MouFlushQue** called.
0x0015	**MouSetDevStatus** called.

Parameter	Description
ulReserved2	Specifies a reserved value that must not be changed. This value represents the return address of the program that calls the specified **Mou** function.
usParam1-usParam5	Specifies up to five values passed with the original **Mou** function call. The actual number and type of parameters used depend on the specific function.

The registered function should return −1 if it wants the original function called, 0 if no error occurred, or an error value.

The entry-point function should determine which function is requested and then carry out an appropriate action using the passed parameters. If necessary, the entry-point function can call a replacement function within the given module to carry out the task. The entry-point or replacement function must leave the stack in the same state it was received. This is required since the return addresses on the stack must be available in the correct order to return control to the program that originally called the **MouRegister** function.

In general, if the replacement function needs to access the mouse, it must use the input-and-output control functions for the mouse. For more information, see Chapter 3, "Input-and-Output Control Functions."

The **MouRegister** function itself cannot be replaced.

See Also **MouDeRegister**

■ MouRemovePtr

```
USHORT MouRemovePtr(pmourtRect, hmou)
PNOPTRRECT pmourtRect;    /* pointer to structure with exclusion rectangle */
HMOU hmou;                /* mouse handle                                  */
```

The **MouRemovePtr** function removes the mouse pointer from a portion of the screen or from the entire screen. This part of the screen is called an exclusion rectangle, because when the mouse pointer moves into it, the pointer disappears—it is still present and can be moved, but it will not appear until it is moved out of the exclusion rectangle. If the pointer is outside the exclusion rectangle and is *not* currently displayed, MS OS/2 draws the mouse pointer.

The **MouRemovePtr** function may be called by any process in the screen group. Only one exclusion rectangle is active at a time, so each call to the function replaces the previous rectangle. The **MouDrawPtr** function removes the exclusion rectangle completely.

Parameters

pmourtRect Points to the **NOPTRRECT** structure that contains the coordinates of the exclusion rectangle. The **NOPTRRECT** structure has the following form:

```
typedef struct _NOPTRRECT {
    USHORT row;
    USHORT col;
    USHORT cRow;
    USHORT cCol;
} NOPTRRECT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE

Comments

You should exclude the mouse pointer from any portion of the screen that is likely to change, such as a text-entry field. When you position the mouse pointer, MS OS/2 saves the character beneath it; when you move the mouse again, MS OS/2 restores the character. If the character changed between the time you positioned the mouse and the time you moved it, the new character is lost when MS OS/2 restores the old character.

Example

This example creates a mouse handle and enables the mouse pointer to be drawn. It then defines an exclusion rectangle in the center of the screen and calls the **MouRemovePtr** function to notify the mouse device driver that this rectangle is for the exclusive use of the process. When you move the mouse pointer into this rectangle, the pointer disappears:

```
NOPTRRECT mourtRect;
HMOU hmou;
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
mourtRect.row = 6;
mourtRect.col = 30;                /* upper-left y-coordinate    */
mourtRect.cRow = 18;               /* lower-right x-coordinate   */
mourtRect.cCol = 50;               /* lower-right y-coordinate   */
MouRemovePtr(&mourtRect, hmou);
```

See Also

MouDrawPtr, MouOpen, MouSetPtrShape

■ MouSetDevStatus

USHORT MouSetDevStatus(*pfsDevStatus, hmou***)**
PUSHORT *pfsDevStatus*; /* pointer to buffer with status */
HMOU *hmou*; /* mouse handle */

The **MouSetDevStatus** function sets the device status for the specified mouse. The device status enables or disables the mouse device driver and defines whether the mouse position is reported in mickeys or in screen units (character cells or pels).

Parameters

pfsDevStatus Points to the variable that contains the device status to be set. This parameter can be any combination of the following values:

Value	Meaning
MOUSE_DISABLED	Disable the default mouse device driver. If this value is not given, the function enables the mouse device driver.
MOUSE_MICKEYS	Report mouse motion in mickeys; that is, MS OS/2 reports motion as a number of mickeys moved from the last-reported position. If the value is not given, MS OS/2 reports mouse motion in screen units relative to the upper-left corner of the screen.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE

Comments

The **MouSetDevStatus** function enables or disables the mouse device driver. When this device driver is enabled, it draws the pointer by combining the AND and XOR masks of the pointer shape with the contents of the screen at the current mouse location. It draws the pointer whenever the mouse moves (or when an interrupt associated with the mouse occurs). When the mouse device driver is disabled, the function does not draw the pointer. In such cases, the process must draw the pointer for itself.

The **MouSetDevStatus** function also directs the mouse to report relative or absolute positions. If the device is set to report absolute positions, the *x*- and *y*-coordinates given for a mouse position are in screen units relative to the upper-left corner of the screen. The type of unit depends on the screen mode. In text mode, the position is given in character cells; in graphics mode, the position is given in pels. Screen coordinates increase from left to right on the *x*-axis and from top to bottom on the *y*-axis. If the device is set to report relative positions, the *x*- and *y*-coordinates for a mouse position are given in mickeys and are relative to the most recently reported position. In this case, the coordinates are signed values, defining both the direction and distance of the move. The *x*-coordinate is negative when the mouse moves left; the *y*-coordinate is negative when the mouse moves up.

Example

This example creates a mouse handle then calls the **MouGetDevStatus** function to set the device status so that mouse-movement information is returned in terms of mickeys, not pels. This allows the process to obtain mouse information in terms of relative movement rather than in terms of absolute pel position:

```
USHORT fsDevStatus = 0x0200;              /* returns mickeys    */
HMOU hmou;
MouOpen(OL, &hmou);
MouSetDevStatus(&fsDevStatus, hmou);      /* sets device status */
```

See Also

MouGetDevStatus, MouOpen

MouSetEventMask

```
USHORT MouSetEventMask(pfsEvents, hmou)
PUSHORT pfsEvents;    /* pointer to buffer with event mask */
HMOU hmou;            /* mouse handle                      */
```

The **MouSetEventMask** function sets the event mask for the specified mouse. The event mask defines the user actions that generate mouse events (movement or pressing or releasing a button).

The **MouSetEventMask** function enables or disables specific user actions. When an action is enabled, MS OS/2 copies a mouse event to the event queue whenever the user carries out the action. When an action is disabled, no mouse event is copied.

Parameters

pfsEvents Points to the variable that contains the event mask. The variable can be any combination of the following values:

Value	Meaning
MOUSE_MOTION	Enable mouse motion with no-buttons-down event.
MOUSE_MOTION_WITH_BN1_DOWN	Enable mouse motion with button-1-down event.

Value	Meaning
MOUSE_BN1_DOWN	Enable button-1-down event.
MOUSE_MOTION_WITH_BN2_DOWN	Enable mouse motion with button-2-down event.
MOUSE_BN2_DOWN	Enable button-2-down event.
MOUSE_MOTION_WITH_BN3_DOWN	Enable mouse motion with button-3-down event.
MOUSE_BN3_DOWN	Enable button-3-down event.

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE

Comments

Button 1 is the left button on the mouse.

Example

This example creates a mouse handle then calls the **MouSetEventMask** function to set the event mask so that only the mouse motion or the pressing of the left button are recognized by the **MouReadEventQue** function:

```
USHORT fsEvents;
HMOU hmou;
MouOpen(OL, &hmou);

/* detect motion and button 1 */

fsEvents = MOUSE_MOTION |
    MOUSE_MOTION_WITH_BN1_DOWN | MOUSE_MOTION_WITH_BN1_DOWN;
MouSetEventMask(&fsEvents, hmou);
```

See Also

MouGetEventMask, MouOpen, MouReadEventQue

■ MouSetPtrPos

```
USHORT MouSetPtrPos(pmouplPosition, hmou)
PPTRLOC pmouplPosition;   /* pointer to structure for new mouse position */
HMOU hmou;                /* mouse handle                                */
```

The **MouSetPtrPos** function sets the current mouse position to the position pointed to by the *pmouplPosition* parameter. If the pointer is visible, the function moves the mouse pointer to the new location on the screen. The new position is always in screen units and is relative to the upper-left corner of the screen.

Parameters

pmouplPosition Points to the **PTRLOC** structure that contains the new mouse position. The **PTRLOC** structure has the following form:

```
typedef struct _PTRLOC {
    USHORT row;
    USHORT col;
} PTRLOC;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE

Comments

MS OS/2 hides the pointer if the new position is in the exclusion rectangle defined by the most recent call to the **MouRemovePtr** function.

Example

This example creates a mouse handle and calls the **MouSetPtrPos** function to initialize the mouse pointer in the upper-left corner of the screen. It then calls the **MouDrawPtr** function to enable the mouse pointer to be drawn:

```
PTRLOC mouplPosition;
HMOU hmou;
MouOpen(0L, &hmou);
mouplPosition.row = 0;                      /* row zero           */
mouplPosition.col = 0;                      /* column zero        */
MouSetPtrPos(&mouplPosition, hmou);         /* sets mouse position */
MouDrawPtr(hmou);
```

See Also

MouDrawPtr, MouGetPtrPos, MouOpen, MouRemovePtr

■ MouSetPtrShape

```
USHORT MouSetPtrShape(pbBuffer, pmoupsInfo, hmou)
PBYTE pbBuffer;           /* pointer to buffer with shape masks    */
PPTRSHAPE pmoupsInfo;     /* pointer to structure with shape info. */
HMOU hmou;                /* mouse handle                          */
```

The **MouSetPtrShape** function sets the AND and XOR masks that define the shape of the mouse pointer for the specified mouse. **MouSetPtrShape** also sets information about the pointer, such as the width and height of masks and the location of the hot spot.

Parameters

pbBuffer Points to the buffer that contains the new masks.

pmoupsInfo Points to the **PTRSHAPE** structure that contains the new pointer information. The **PTRSHAPE** structure has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_MOUSE_INV_PARMS
ERROR_MOUSE_NO_DEVICE

Comments

The **MouSetPtrShape** function copies the AND and XOR masks from the buffer pointed to by the *pbBuffer* parameter. The format and size of the masks depend on the display device and the video mode. In text mode, each mask is typically a character/attribute pair. In graphics mode, each mask is a bitmap.

The **MouSetPtrShape** function copies information about the pointer from the structure pointed to by the *pmoupsInfo* parameter. The structure defines the length (in bytes) of the AND and XOR masks, the width and height of each mask, and the offset from the current mouse position (or hot spot) to the upper-left corner of the pointer.

If the pointer is displayed, the **MouSetPtrShape** function may not display a new shape immediately. If the pointer is not displayed, you must use the **MouRemovePtr** and **MouDrawPtr** functions to display the new shape.

The pointer shape is dependent on the device driver used to support the display device. In text mode, MS OS/2 supports the pointer shape as a reverse block character. This character has a one-character height and width; that is, in text modes, the height and width fields must each be one. You can determine the current pointer shape in effect for the screen group by using the **MouGetPtrShape** function.

See Also

MouDrawPtr, MouGetPtrShape, MouOpen, MouRemovePtr

■ MouSetScaleFact

USHORT MouSetScaleFact(*pmouscFactors*, *hmou***)**
PSCALEFACT *pmouscFactors*; /* pointer to structure for scaling factors */
HMOU *hmou*; /* mouse handle */

The **MouSetScaleFact** function sets the horizontal and vertical scaling factors for the specified mouse. The scaling factors define the number of mickeys the mouse must travel horizontally or vertically to cause MS OS/2 to move the mouse pointer one screen unit.

Parameters

pmouscFactors Points to the **SCALEFACT** structure that contains the scaling factors. The **SCALEFACT** structure has the following form:

```
typedef struct _SCALEFACT {
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hmou Identifies the mouse. The handle must have been created previously by using the **MouOpen** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle, enables the mouse pointer to be drawn, and retrieves the current scaling factor. It then doubles the scaling factor and calls the **MouSetScaleFact** function to set the new factor. The result is that you must move the mouse twice as far in order to move the pointer on the screen:

```
SCALEFACT mouscFactors;
HMOU hmou;
MouOpen(0L, &hmou);
MouDrawPtr(hmou);
MouGetScaleFact(&mouscFactors,     /* retrieves scaling factors */
    hmou);
mouscFactors.rowScale *= 2;        /* vertical scaling factor   */
mouscFactors.colScale *= 2;        /* horizontal scaling factor */
MouSetScaleFact(&mouscFactors,     /* sets new scaling factors  */
    hmou);
```

See Also

MouGetScaleFact, MouOpen

■ MouSynch

USHORT MouSynch(*fWait*)
USHORT *fWait*; /* wait/no-wait flag */

The **MouSynch** function synchronizes access to the mouse. This function should be used by a **Mou** subsystem to prevent more than one process from accessing the mouse handle at any one time.

Parameters

fWait Specifies whether to wait if the mouse device driver is currently busy. If this parameter is FALSE, the function returns control immediately without waiting for the device to become free. If the parameter is TRUE, the function waits until the mouse handle is free.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Comments

The **MouSynch** function requests an exclusive system semaphore that clears when the **Mou** subsystem returns to the mouse router. The **MouSynch** function blocks all other threads within a screen group until the semaphore clears.

See Also

DosCloseSem, DosDevIOCtl, MouRegister

VioAssociate

```
USHORT VioAssociate(hdc, hvps)
HDC hdc;        /* device-context handle     */
HVPS hvps;      /* presentation-space handle */
```

The **VioAssociate** function associates an advanced video-input-and-output (AVIO) presentation space with a device context. Subsequent calls to the **VioShowPS** and **VioShowBuf** functions direct output to this device context.

A screen device context is the only kind of device context that may be associated with an AVIO presentation space. If the AVIO presentation space is currently associated with another device context, it is disassociated. Similarly, if another AVIO presentation space is currently associated with the device context, it too is disassociated.

If you specify a NULL handle for the device context, the presentation space is disassociated from the currently associated device context.

Parameters

hdc Identifies the device context to associate with the presentation space. If this parameter is NULL, the function disasssociates the previous device context.

hvps Identifies the AVIO presentation space. The space must have been created previously by using the **VioCreatePS** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

VioCreatePS, VioShowBuf, VioShowPS, WinOpenWindowDC

VioCreateLogFont

```
USHORT VioCreateLogFont(pfat, lcid, pstr8Name, hvps)
PFATTRS pfat;       /* pointer to structure for font attributes    */
LONG lcid;          /* local identifier for font                   */
PSTR8 pstr8Name;    /* pointer to descriptive name of logical font */
HVPS hvps;          /* presentation-space handle                   */
```

The **VioCreateLogFont** function creates a logical font for the given advanced video-input-and-output (AVIO) presentation space. A logical font is a list of attributes, such as character size and weight, that specifies the font used for writing text. When a font is needed, MS OS/2 chooses from the available physical fonts the one that most closely matches the logical font. A program may, however, force selection of a particular font by setting the **lMatch** field in the **FATTRS** structure to the value returned for the requested font by the **VioQueryFonts** function.

If the **szFaceName** field in the **FATTRS** structure is NULL and all of the attributes except the code page are set to zero, the system default font is selected, in the specified code page.

Parameters

pfat Points to the **FATTRS** structure that contains the attributes of the font. The **FATTRS** structure has the following form:

```
typedef struct _FATTRS {
   USHORT usRecordLength;
   USHORT fsSelection;
   LONG   lMatch;
   CHAR   szFaceName[FACESIZE];
   USHORT idRegistry;
   USHORT usCodePage;
   LONG   lMaxBaselineExt;
   LONG   lAveCharWidth;
   USHORT usWidthClass;
   USHORT usWeightClass;
   USHORT fsType;
   SHORT  sQuality;
   USHORT fsFontUse;
} FATTRS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

lcid Specifies the local identifier for the font. This parameter must be 1, 2, or 3. If the identifier is already being used to refer to a font or bitmap, the function returns an error.

pstr8Name Points to an 8-character name that you may use to describe the logical font.

hvps Identifies the AVIO presentation space. This presentation space must have been created previously by using the **VioCreatePS** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, indicating that an error occurred.

See Also

VioQueryFonts

VioCreatePS

USHORT VioCreatePS(*phvps, cRows, cColumns, fFormat, cAttrBytes, hvps***)**

PHVPS *phvps*;	/* pointer to variable for presentation-space handle */
SHORT *cRows*;	/* height of presentation space */
SHORT *cColumns*;	/* width of presentation space */
SHORT *fFormat*;	/* format of attribute byte(s) */
SHORT *cAttrBytes*;	/* number of attributes */
HVPS *hvps*;	/* presentation-space handle */

The **VioCreatePS** function creates an advanced video-input-and-output (AVIO) presentation space, the size of which must not exceed 32K. To determine the size of the presentation space, multiply the *cColumns*, *cRows*, and *cAttrBytes* parameters as follows: $cColumns \times cRows \times (cAttrBytes + 1)$.

Parameters

phvps Points to the variable that receives the presentation-space handle. You may use this handle in subsequent **Vio** functions.

cRows Specifies the height (in character cells) of the presentation space.

cColumns Specifies the width (in character cells) of the presentation space.

fFormat Identifies the format of the attribute byte(s) in the presentation space. The content of the attribute bytes depends on the format. Currently, the only defined format is zero. If the format is zero, the attribute bytes have the following meanings:

Value	Meaning
FORMAT_CGA	Specifies a CGA format of two attribute bytes. The first byte contains the character value. The second byte contains bit fields that specify the background and foreground colors. Blink and intensity fields are not supported.
FORMAT_4BYTE	Specifies an extended format of four attribute bytes. The first byte contains the character value. The second byte contains bit fields that specify the background and foreground colors. The third byte contains bit fields that specify the underscore, reverse video, the background opacity, and the font identifier. The fourth byte is an extra byte to be used by programs.

cAttrBytes Specifies the number of attribute bytes per character cell in the presentation space. This number may be 1 or 3.

hvps Identifies the AVIO presentation space. This parameter must be zero.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

See Also **VioDestroyPS**

VioDeleteSetId

```
USHORT VioDeleteSetId(lcid, hvps)
LONG lcid;      /* local identifier for object   */
HVPS hvps;      /* presentation-space handle */
```

The **VioDeleteSetId** function deletes the logical font specified by the *lcid* parameter. Do not use this function to delete the object specified by the local identifier zero.

Parameters

lcid Specifies the local identifier for the object. This parameter must be 1, 2, or 3. If you specify -1, this function deletes all logical fonts.

hvps Identifies the advanced video-input-and-output (AVIO) presentation space. This presentation space must have been created previously by using the **VioCreatePS** function.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

See Also **VioCreateLogFont, VioCreatePS**

VioDeRegister

USHORT VioDeRegister(*VOID* **)**

The **VioDeRegister** function restores the functions of the default **Vio** subsystem and releases any previously registered **Vio** subsystem. The function restores the default **Vio** subsystem for all processes in the current screen group.

Once a process registers a **Vio** subsystem, no other process in the screen group may register a **Vio** subsystem until the default subsystem is restored. Only the process registering a **Vio** subsystem may call the **VioDeRegister** function to restore the default **Vio** subsystem.

Parameters This function has no parameters.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_DEREGISTER

See Also **VioRegister**

VioDestroyPS

USHORT VioDestroyPS(*hvps* **)**
HVPS *hvps*; /* presentation-space handle */

The **VioDestroyPS** function destroys the specified advanced video-input-and-output (AVIO) presentation space.

Parameters *hvps* Identifies the AVIO presentation space to destroy. This presentation space must have been created previously by using the **VioCreatePS** function.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

See Also **VioCreatePS**

VioEndPopUp

USHORT VioEndPopUp(*hvio* **)**
HVIO *hvio*; /* video handle */

The **VioEndPopUp** function closes a pop-up screen and restores the physical video buffer to its previous contents. Only the process that opened the pop-up screen may close it.

VioEndPopUp may not completely restore the screen to its previous state. For example, programs that modify the video registers or use graphics modes may have to restore the state of the registers as the pop-up screen is being closed. By calling the **VioModeWait** function, a program can request to be notified of the change in video mode. Whenever a process has a pending request, MS OS/2 notifies the process of a mode change when the pop-up screen is closed.

Parameters

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_NO_POPUP

Example

This example creates a pop-up screen, displays a message, waits three seconds, then calls **VioEndPopUp** to close the pop-up screen:

```
USHORT fWait = VP_WAIT;
VioPopUp(&fWait, 0);                    /* creates pop-up screen */
VioWrtTTY("This is a VIO pop-up screen\n\r", 29, 0);
DosSleep(3000L);                        /* waits 3 seconds       */
VioEndPopUp(0);                         /* ends pop-up screen    */
```

See Also

VioModeWait, VioPopUp

VioGetAnsi

USHORT VioGetAnsi(*pfAnsi*, *hvio*)

```
PUSHORT pfAnsi;    /* pointer to variable for ANSI flag */
HVIO hvio;         /* video handle                      */
```

The **VioGetAnsi** function retrieves the state of the ANSI flag, which determines whether the processing of ANSI escape sequences is enabled or disabled.

Parameters

pfAnsi Points to the variable that receives the ANSI flag. If this flag is ANSI_ON, ANSI processing is enabled. If the flag is ANSI_OFF, ANSI processing is disabled.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example calls **VioGetAnsi** and, if ANSI processing enabled, calls the VioWrtTTY function to display a message:

```
USHORT fAnsi;
VioGetAnsi(&fAnsi, 0);
if (fAnsi == ANSI_ON)
    VioWrtTTY("ANSI is on\n\r", 12, 0);
```

See Also

VioSetAnsi, VioWrtTTY

■ VioGetBuf

USHORT VioGetBuf(*pulLVB, pcbLVB, hvio***)**

```
PULONG pulLVB;      /* pointer to variable for address of LVB */
PUSHORT pcbLVB;     /* pointer to variable for length of LVB  */
HVIO hvio;          /* video handle                           */
```

The **VioGetBuf** function retrieves the address of the logical video buffer (LVB), which contains the current character attributes for the text output of a process. The logical video buffer is identical in content and format to the physical video buffer when the process is the foreground process. The logical video buffer is available for text-mode screens only.

A process can access and modify the contents of the logical video buffer at any time, even if the process is in the background. Changes made to the logical video buffer do not affect the physical screen until the process calls the **VioShowBuf** function.

The **VioGetBuf** function is a family API function.

Parameters

pulLVB Points to the variable that receives the address of the logical video buffer.

pcbLVB Points to the variable that specifies the length (in bytes) of the logical video buffer. You can use the **VioGetMode** function to determine the dimensions of the buffer.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example calls **VioGetBuf** to retrieve the address of the logical video buffer. It sets the character attributes in the buffer for foreground blinking by using the OR operator to set the high bit, then it calls the **VioShowBuf** function to display the character attributes:

```
PBYTE pbLVB;
USHORT cbLVB, i;
VioGetBuf((PULONG) &pbLVB, &cbLVB, 0);
for (i = 0; i < cbLVB; i += 2)

    /* OR in the high bit to make it a blinking attribute */

    *(pbLVB + i + 1) = *(pbLVB + i + 1) | 0x80;
VioShowBuf(0, cbLVB, 0);            /* displays buffer     */
```

See Also

VioGetMode, VioGetPhysBuf, VioShowBuf

VioGetConfig

```
USHORT VioGetConfig(usReserved, pvioin, hvio)
USHORT usReserved;          /* must be zero                        */
PVIOCONFIGINFO pvioin;      /* pointer to structure for configuration */
HVIO hvio;                  /* video handle                        */
```

The **VioGetConfig** function retrieves the video display configuration, which defines the type of display adapter, the type of display, and the amount of video memory available.

The **VioGetConfig** function is a family API function.

Parameters

usReserved Specifies a reserved value. This parameter must be zero.

pvioin Points to the **VIOCONFIGINFO** structure that receives the display configuration for the primary display adapter. The **VIOCONFIGINFO** structure has the following form:

```
typedef struct _VIOCONFIGINFO {
    USHORT cb;
    USHORT adapter;
    USHORT display;
    ULONG  cbMemory;
} VIOCONFIGINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_LENGTH
ERROR_VIO_INVALID_PARMS

Comments

MS OS/2 derives the values for the **adapter** and **display** fields for the display configuration by using various tests, including checking the switch settings on the card.

Example

This example calls **VioGetConfig** to determine whether the display type is an enhanced color display:

```
VIOCONFIGINFO vioinConfig;
vioinConfig.cb = sizeof(vioinConfig);      /* structure length   */
VioGetConfig(0,                            /* must be zero       */
    &vioinConfig,                          /* configuration data */
    0);                                    /* video handle       */
if (vioinConfig.display == 2)
    VioWrtTTY("Enhanced color display\n\r", 24, 0);
```

See Also

VioGetMode, VioGetState

■ VioGetCp

```
USHORT VioGetCp(usReserved, pIdCodePage, hvio)
USHORT usReserved;        /* must be zero                  */
PUSHORT pIdCodePage;      /* pointer to code-page identifier */
HVIO hvio;                /* video handle                  */
```

The **VioGetCp** function retrieves the identifier of the code page for the current screen group. This code page defines the character set being used to display text on the screen. If the identifier is 0x0000, the system default code page is being used. Any other value identifies a code page that has been set by using the **VioSetCp** function or that has been inherited from the parent process.

Parameters

usReserved Specifies a reserved value. This parameter must be zero.

pIdCodePage Points to the variable that receives the code-page identifier. The following are the valid code-page numbers:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example calls **VioGetCp** to retrieve the current system code page:

```
USHORT idCodePage;
VioGetCp(0,                /* must be zero          */
    &idCodePage,           /* code-page identifier */
    0);                    /* video handle          */
```

See Also

DosGetCp, DosSetCp, VioSetCp

■ VioGetCurPos

```
USHORT VioGetCurPos(pusRow, pusColumn, hvio)
PUSHORT pusRow;        /* pointer to variable for row    */
PUSHORT pusColumn;     /* pointer to variable for column */
HVIO hvio;             /* video handle                   */
```

The **VioGetCurPos** function retrieves the position of the cursor on the screen.

The **VioGetCurPos** function is a family API function.

Parameters

pusRow Points to the variable that receives the current row position of the cursor.

pusColumn Points to the variable that receives the current column position of the cursor.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example calls **VioGetCurPos** to retrieve the current row-and-column position of the cursor:

```
USHORT usRow, usColumn;
VioGetCurPos(&usRow,          /* row address    */
    &usColumn,                /* column address */
    0);                       /* video handle   */
```

See Also

VioGetCurType, VioSetCurPos

■ VioGetCurType

USHORT VioGetCurType (*pviociCursor, hvio***)**
PVIOCURSORINFO *pviociCursor*; /* pointer to structure for cursor info */
HVIO *hvio*; /* video handle */

The **VioGetCurType** function retrieves information about the cursor type. This information defines the height and width of the cursor, as well as whether it is currently visible. The **VioGetCurType** function is a family API function.

Parameters

pviociCursor Points to the **VIOCURSORINFO** structure that receives information about the cursor type. The **VIOCURSORINFO** structure has the following form:

```
typedef struct _VIOCURSORINFO {
    USHORT yStart;
    USHORT cEnd;
    USHORT cx;
    USHORT attr;
} VIOCURSORINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example calls **VioGetCurType** to retrieve the current cursor type, changes the attribute to hidden or visible (the opposite of what it was), and calls **VioSetCurType** to set the new cursor type:

```
VIOCURSORINFO viociCursor;
VioGetCurType(&viociCursor, 0); /* retrieves current cursor type     */
viociCursor.attr =              /* flips attribute to hidden/visible */
    (viociCursor.attr == -1) ? 0 : -1;
VioSetCurType(&viociCursor, 0); /* sets new cursor type              */
```

See Also

VioGetCurPos, VioSetCurType

VioGetDeviceCellSize

```
USHORT VioGetDeviceCellSize(pcRows, pcColumns, hvps)
PSHORT pcRows;       /* pointer to variable for cell height */
PSHORT pcColumns;    /* pointer to variable for cell width  */
HVPS hvps;           /* presentation-space handle           */
```

The **VioGetDeviceCellSize** function retrieves the size of the current device cell.

Parameters

pcRows Points to the variable that specifies the height (in pels) of the device cell.

pcColumns Points to the variable that specifies the width (in pels) of the device cell.

hvps Identifies the advanced video-input-and-output (AVIO) presentation space. This presentation space must have been created previously by using the **VioCreatePS** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

VioCreatePS, VioSetDeviceCellSize

VioGetFont

```
USHORT VioGetFont(pviofi, hvio)
PVIOFONTINFO pviofi;   /* pointer to structure for font information */
HVIO hvio;             /* video handle                              */
```

The **VioGetFont** function retrieves a specified font. A font consists of one bitmap for each character in a character set. The bitmaps define the character shapes. The **VioGetFont** function retrieves a copy of either the current font or a font from the ROM of the video display adapter.

Parameters

pviofi Points to the **VIOFONTINFO** structure that specifies the request type and receives the font information. The **VIOFONTINFO** structure has the following form:

```
typedef struct _VIOFONTINFO {
    USHORT cb;
    USHORT type;
    USHORT cxCell;
    USHORT cyCell;
    PVOID pbData;
    USHORT cbData;
} VIOFONTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_FONT
ERROR_VIO_INVALID_PARMS
ERROR_VIO_ROW

Comments

Although the **VioGetFont** function can retrieve fonts for many display adapters, the fonts for some adapters are not available. In most cases, the function retrieves a full 256-character font. This font may consist of a complete ROM font, or it may be derived from downloaded fonts that are saved in the adapter's BIOS. The current font is defined by the most recent **DosSetCp** or **VioSetCp** function, or it can be set by using the **VioSetFont** function.

Example

This example calls the **VioGetFont** function to obtain the current font. When it returns, the **cxCell** and **cyCell** fields will contain the dimensions (in points) of a character cell. The **pbData** field points to the font:

```
VIOFONTINFO viofiFont;
viofiFont.cb = sizeof(viofiFont);           /* length of structure    */
viofiFont.type = VGFI_GETCURFONT;           /* retrieves current font */
viofiFont.cxCell = 0;                       /* clears columns         */
viofiFont.cyCell = 0;                       /* clears rows            */
viofiFont.pbData = 0L;                      /* address of data area   */
viofiFont.cbData = 0;                       /* length of data area    */
VioGetFont(&viofiFont, 0);
```

See Also

DosSetCp, VioSetCp, VioSetFont

■ VioGetMode

```
USHORT VioGetMode(pviomi, hvio)
PVIOMODEINFO pviomi;    /* pointer to structure for screen mode information */
HVIO hvio;              /* video handle                                     */
```

The **VioGetMode** function retrieves the current screen mode. The screen mode defines the display mode (text or graphics), the number of colors being used (2, 4, or 16), and the width and height of the screen in both character cells and pels.

The **VioGetMode** function is a family API function.

Parameters

pviomi Points to the **VIOMODEINFO** structure that receives the screen-mode information. The **VIOMODEINFO** structure has the following form:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
} VIOMODEINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH

Example

This example calls **VioGetMode** to retrieve the mode information for the screen:

```
VIOMODEINFO viomi;
viomi.cb = sizeof(viomi);
VioGetMode(&viomi, 0);
if (viomi.fbType == 0)
    VioWrtTTY("Monochrome display\n\r", 20, 0);
```

See Also

VioGetState, VioSetMode

■ VioGetOrg

```
USHORT VioGetOrg(psRow, psColumn, hvps)
PSHORT psRow;       /* pointer to variable for row number    */
PSHORT psColumn;    /* pointer to variable for column number */
HVPS hvps;          /* presentation-space handle             */
```

The **VioGetOrg** function retrieves the origin of an advanced video-input-and-output (AVIO) presentation space.

Parameters

psRow Points to the variable that receives the row number of the cell currently mapped to the upper-left corner of the window.

psColumn Points to the variable that receives the column number of the cell currently mapped to the upper-left corner of the window.

hvps Identifies the AVIO presentation space. This presentation space must have been created previously by using the **VioCreatePS** function

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

VioCreatePS, VioSetOrg

■ VioGetPhysBuf

USHORT VioGetPhysBuf(*pviopb, usReserved***)**
PVIOPHYSBUF *pviopb*; /* pointer to structure for physical video buffer */
USHORT *usReserved*; /* must be zero */

The **VioGetPhysBuf** function retrieves the selector of the physical video buffer. The physical video buffer contains the text or graphics information that defines the current screen image. In text mode, the buffer contains the character and attribute for each character cell. In graphics mode, the buffer is a bitmap (in one or more planes) of the image on the screen. The content of the screen depends on the current screen mode and the type of display adapter.

The **VioGetPhysBuf** function is a family API function.

Parameters

pviopb Points to the **VIOPHYSBUF** structure that specifies the address and length of the physical video buffer, and receives the selector(s) used to address the video buffer. The **VIOPHYSBUF** structure has the following form:

```
typedef struct _VIOPHYSBUF {
    PBYTE pBuf;
    ULONG cb;
    SEL   asel[1];
} VIOPHYSBUF;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

usReserved Specifies a reserved value. This parameter must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_IN_BG
ERROR_VIO_INVALID_HANDLE

Comments

Since the physical video buffer is subject to change by the current foreground process, only the foreground process should access the buffer. To ensure that the foreground process has complete control of the physical buffer, use the **VioScrLock** function.

Example

This example locks the screen, calls **VioGetPhysBuf** to retrieve the address of the physical video buffer, unlocks the screen, and assigns the address of the physical video buffer to a pointer:

```
VIOPHYSBUF viopbBuffer;
PCH pchScreen;
USHORT fStatus;
viopbBuf.pBuf = 0xB8000L;
viopbBuf.cb = 4000;
VioScrLock(LOCKIO_WAIT, &fStatus, 0);
VioGetPhysBuf(&viopbBuf, 0);
VioScrUnLock(0);
pchScreen = MAKEP(viopbBuf.asel[0]), 0);
```

See Also

VioGetBuf, VioScrLock, VioScrUnLock, VioShowBuf

VioGetState

USHORT VioGetState(*pvoidState*, *hvio*)
PVOID *pvoidState*; /* pointer to structure for state information */
HVIO *hvio*; /* video handle */

The **VioGetState** function retrieves the current settings of the palette registers, the overscan (border) color, or the blink/background intensity switch.

Parameters

pvoidState Points to the structure that receives the state information. The structure type, which depends on the request type specified in the **type** field of each structure, is one of the following: **VIOPALSTATE**, **VIOOVERSCAN**, or **VIOINTENSITY**. These structures have the following forms:

```
typedef struct _VIOPALSTATE {
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;

typedef struct _VIOOVERSCAN {
    USHORT cb;
    USHORT type;
    USHORT color;
} VIOOVERSCAN;

typedef struct _VIOINTENSITY {
    USHORT cb;
    USHORT type;
    USHORT fs;
} VIOINTENSITY;
```

For each structure, you must set the **cb** and **type** fields before calling the function. Not all values for the **type** field are valid for all screen modes.

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH

Example

This example calls the **VioGetState** function to retrieve the settings for each of the 16 palette registers:

```
BYTE abState[38];
PVIOPALSTATE pviopal;
pviopal = (PVIOPALSTATE) abState;
pviopal->cb = sizeof(abState);    /* structure size              */
pviopal->type = 0;                /* retrieves palette registers */
pviopal->iFirst = 0;              /* first palette register to return */
VioGetState(pviopal, 0);
```

See Also

VioGetMode, VioSetState

VioModeUndo

```
USHORT VioModeUndo(fRelinquish, fTerminate, hvio)
USHORT fRelinquish;    /* ownership flag   */
USHORT fTerminate;     /* termination flag */
USHORT hvio;           /* video handle     */
```

The **VioModeUndo** function cancels a request by a process to be notified of a change in video mode. A process makes this request by calling the **VioModeWait** function. The request forces the calling thread to wait until the video mode changes. The **VioModeUndo** function cancels the request and permits the thread to continue (or ends the thread, if requested to do so).

MS OS/2 permits only one process in a screen group to request notification of a video-mode change. The first process to make a request owns it. Thereafter, other processes must wait for the owning process to relinquish the request before being granted ownership. To force a process to relinquish ownership of the request, use the **VioModeUndo** function.

Only the process that owns the change-mode request may call the **VioModeUndo** function.

Parameters

fRelinquish Specifies whether the process should retain or relinquish ownership of the request. If this parameter is UNDOI_GETOWNER, the process retains ownership and can make the request again without competing with other processes. If this parameter is UNDOI_RELEASEOWNER, the process relinquishes ownership of the request and is canceled by **VioModeUndo**.

fTerminate Specifies whether to terminate the thread waiting for the mode change. If this parameter is UNDOK_ERRORCODE, the thread continues and receives an error value from the **VioModeWait** function. If the parameter is UNDOK_TERMINATE, the thread terminates.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_FUNCTION_OWNED
ERROR_VIO_INVALID_PARMS
ERROR_VIO_NO_MODE_THREAD

See Also

VioModeWait

VioModeWait

```
USHORT VioModeWait(fEvent, pfNotify, hvio)
USHORT fEvent;       /* event flag                          */
PUSHORT pfNotify;    /* pointer to variable for notify flag */
USHORT hvio;         /* video handle                        */
```

The **VioModeWait** function waits for a change in the current video mode before returning. When a change occurs, MS OS/2 sets the variable pointed to by the *pfNotify* parameter to a value indicating the type of change. The thread may then restore the video registers or carry out other tasks related to restoring the video mode for the process.

The **VioModeWait** function is used typically by graphics programs (or text programs that access video registers directly) to restore the screen after a pop-up screen has closed. Pop-up screens often change the video mode and video-register values without fully restoring them when closed. A thread that calls the **VioModeWait** function waits until a pop-up screen closes so that it can restore the screen.

MS OS/2 permits only one process in a screen group to wait for a video-mode change. The first process to make a request owns it.

Parameters

fEvent Specifies the event flag of the event to wait for. If this parameter is VMWR_POPUP, the function waits for a pop-up screen to close. No other flags are permitted.

pfNotify Points to the variable that receives a flag specifying the action to carry out in response to the given event. If this flag is VMWN_POPUP, the process should restore the video mode. No other values are returned.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_FUNCTION_OWNED
ERROR_VIO_INVALID_PARMS
ERROR_VIO_RETURN

Comments

A program should use the **VioModeWait** function if it changes the video registers directly. MS OS/2 automatically saves and restores the physical video buffer and screen mode whenever a pop-up screen is used.

The thread that calls **VioModeWait** should carry out only those tasks directly related to restoring the screen mode. Whenever a mode change occurs, the thread should restore the mode and call **VioModeWait** as quickly as possible. The thread should not call MS OS/2 functions (netiher directly nor indirectly through other functions) that may generate pop-up screens or error pop-up screens. Doing so may cause MS OS/2 to lock up (that is, each call of the thread generates a pop-up screen, which in turn calls the thread and generates another pop-up screen, and so on). You can use the **VioModeUndo** function to end the thread when it is no longer needed.

Programs that save and restore the video mode and screen before and after a screen switch should use the **VioSaveRedrawWait** function.

See Also

VioModeUndo, VioPopUp, VioSaveRedrawWait

VioPopUp

```
USHORT VioPopUp(pfWait, hvio)
PUSHORT pfWait;    /* pointer to variable for wait/no-wait flag */
HVIO hvio;         /* video handle                              */
```

The **VioPopUp** function opens a pop-up screen. A pop-up screen is a temporary text-mode screen that a process can use to display error and warning messages without altering the content of the foreground screen. Pop-up screens are used typically by background processes to display messages when the screen is not available.

The pop-up screen can be opaque or transparent, as specified by the flag pointed to by the *pfWait* parameter. If the pop-up screen is opaque, the function changes the screen mode (if the mode is not already set for 25 lines by 80 columns of text) and clears the screen, moving the cursor to the upper-left corner. If the pop-up screen is transparent, the function uses the current screen mode and leaves the screen and the cursor unchanged.

Once the pop-up screen is open, the process may call any of the following **Vio** functions:

VioEndPopUp	**VioReadCellStr**	**VioSetFont**
VioGetAnsi	**VioReadCharStr**	**VioSetState**
VioGetCp	**VioScrollDn**	**VioWrtCellStr**
VioGetConfig	**VioScrollLf**	**VioWrtCharStr**
VioGetCurPos	**VioScrollRt**	**VioWrtCharStrAtt**
VioGetCurType	**VioScrollUp**	**VioWrtNAttr**
VioGetFont	**VioSetCp**	**VioWrtNCell**
VioGetMode	**VioSetCurPos**	**VioWrtNChar**
VioGetState	**VioSetCurType**	**VioWrtTTY**

The process opening the pop-up screen receives all subsequent keyboard input, and MS OS/2 disables the keys that it normally uses switch from one screen group to another. While the pop-up screen is open, the process must not access or modify the physical video buffer. Also, it must not call the **DosExecPgm** function.

Only one pop-up screen may be open at any given time. If a process attempts to open one pop-up screen while another is already open, the **VioPopUp** function waits until the previous screen is closed before opening the new one.

Parameters

pfWait Points to the variable that specifies whether the pop-up screen is to be opaque or transparent, and whether the function should wait for any open pop-up screen to close. It can be any combination of either VP_NOWAIT or VP_WAIT and either VP_OPAQUE or VP_TRANSPARENT. These flags are defined as follows:

Value	Meaning
VP_NOWAIT	Return immediately if a pop-up screen already exists.
VP_WAIT	Wait if a pop-up screen already exists. The function opens a new pop-up screen as soon as the existing one is closed.

Value	Meaning
VP_OPAQUE	Set the screen mode for 25 lines by 80 columns of text, clear the screen, and move the cursor to the upper-left corner.
VP_TRANSPARENT	Create a transparent pop-up screen. The function does not change the screen mode, clear the screen, or move the cursor. To create a transparent pop-up screen, the screen must be in text mode already.

hvio Identifies a reserved value. This parameter must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_EXISTING_POPUP
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_NO_POPUP

Comments

Before opening a pop-up screen, MS OS/2 saves the physical video buffer of the existing screen. While the pop-up screen is open, MS OS/2 blocks any **Vio** functions called by the process that owns the previous screen. If this process generates any output, MS OS/2 displays the output *after* the pop-up screen closes.

You can close a pop-up screen by using the **VioEndPopUp** function. **VioEndPopUp** restores the screen mode and the screen buffer; it also restores keyboard input to the previous process and enables the key combination MS OS/2 uses to switch screen groups. In some cases, the **VioEndPopUp** function may not completely restore the screen. For these cases, use the **VioModeWait** function to restore the screen.

You cannot use transparent pop-up screens if the foreground process has called the **VioSavRedrawWait** function.

If a process registers a replacement **VioPopUp** function (by calling the **VioRegister** function), MS OS/2 uses the replacement function only if the foreground process requests a pop-up screen. If a background process requests a pop-up screen, MS OS/2 uses the default **VioPopUp** function.

Example

This example calls the **VioPopUp** function to create a pop-up screen, and waits for the pop-up screen if another pop-up screen is already active:

```
USHORT fWait = VP_WAIT | VP_OPAQUE;
VioPopUp(&fWait, 0);
    .
    . /* message and user interaction would go here */
    .
VioEndPopUp(0);    /* ends pop-up screen */
```

See Also

DosExecPgm, VioEndPopUp, VioGetPhysBuf, VioModeWait, VioRegister, VioSavRedrawWait

VioPrtSc

USHORT VioPrtSc(*hvio***)**
HVIO *hvio*; /* video handle */

The **VioPrtSc** function copies the contents of the screen to the printer.

This function is reserved for system use. It is called whenever the PRINTSCREEN key is pressed. A process can, however, replace **VioPrtSc** with a custom screen-printing function by using the **VioRegister** function. If a process does replace the **VioPrtSc** function, all other processes in the screen group will also use the replacement function. This gives a process the capability of capturing input from the PRINTSCREEN key.

Parameters

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_SMG_ONLY

See Also

VioPrtScToggle, VioRegister

VioPrtScToggle

USHORT VioPrtScToggle(*hvio***)**
HVIO *hvio*; /* video handle */

The **VioPrtScToggle** function enables or disables the printer echo feature.

This function is reserved for system use. It is called whenever the CTRL+PRTSC key combination is pressed. The first press enables the printer echo feature, the second disables it. A process can replace **VioPrtScToggle**, however, with a custom function by using the **VioRegister** function. If a process does replace the **VioPrtScToggle** function, all processes in the screen group will also use the replacement function. This gives a process the capability of capturing input from the CTRL+PRTSC key combination.

Parameters

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_SMG_ONLY

See Also

VioPrtSc, VioRegister

VioQueryFonts

```
USHORT VioQueryFonts(pcbMetrics, pfm, cbMetrics, pcFonts, pszFacename, flOptions, hvps)
PLONG pcbMetrics;           /* pointer to variable for structure length */
PFONTMETRICS pfm;           /* pointer to structure for font metrics    */
LONG cbMetrics;             /* length of structure                      */
PLONG pcFonts;              /* pointer to variable for number of fonts  */
PSZ pszFacename;            /* pointer to string for face name          */
ULONG flOptions;            /* enumeration options                      */
HVPS hvps;                  /* presentation-space handle                */
```

The **VioQueryFonts** function retrieves a font-metrics structure (or structures) that contains characteristics of the fonts that match the specified face name. These characteristics, or font metrics, are returned for as many matching fonts as will fit in the structure pointed to by the *pfm* parameter.

After examining the returned data, the application selects the font most appropriate for its requirements, and if necessary, forces selection of a particular font by specifying the **lMatch** field (as returned in the *pfm* parameter) in the **FATTRS** structure for the **VioCreateLogFont** function.

By specifying zero for the *pcFonts* parameter and then examining the value returned, the application determines how many fonts match the specified face name.

All sizes are returned in world coordinates. For more information, see the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

Parameters

pcbMetrics Points to the variable that receives the length (in bytes) of each **FONTMETRICS** structure. The structure pointed to by the *pfm* parameter must contain the number of bytes given by *pcFonts* × *pcMetrics*.

pfm Points to the **FONTMETRICS** structure that receives the font metrics of the specified matching fonts. The format for each record is as defined in the **GpiQueryFontMetrics** function. The **FONTMETRICS** structure has the following form:

```
typedef struct _FONTMETRICS {
    CHAR    szFamilyname[FACESIZE];
    CHAR    szFacename[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lEmHeight;
    LONG    lXHeight;
    LONG    lMaxAscender;
    LONG    lMaxDescender;
    LONG    lLowerCaseAscent;
    LONG    lLowerCaseDescent;
    LONG    lInternalLeading;
    LONG    lExternalLeading;
    LONG    lAveCharWidth;
    LONG    lMaxCharInc;
    LONG    lEmInc;
    LONG    lMaxBaselineExt;
    SHORT   sCharSlope;
    SHORT   sInlineDir;
    SHORT   sCharRot;
    USHORT  usWeightClass;
    USHORT  usWidthClass;
    SHORT   sXDeviceRes;
    SHORT   sYDeviceRes;
    SHORT   sFirstChar;
    SHORT   sLastChar;
```

```
    SHORT   sDefaultChar;
    SHORT   sBreakChar;
    SHORT   sNominalPointSize;
    SHORT   sMinimumPointSize;
    SHORT   sMaximumPointSize;
    USHORT  fsType;
    USHORT  fsDefn;
    USHORT  fsSelection;
    USHORT  fsCapabilities;
    LONG    lSubscriptXSize;
    LONG    lSubscriptYSize;
    LONG    lSubscriptXOffset;
    LONG    lSubscriptYOffset;
    LONG    lSuperscriptXSize;
    LONG    lSuperscriptYSize;
    LONG    lSuperscriptXOffset;
    LONG    lSuperscriptYOffset;
    LONG    lUnderscoreSize;
    LONG    lUnderscorePosition;
    LONG    lStrikeoutSize;
    LONG    lStrikeoutPosition;
    SHORT   sKerningPairs;
    SHORT   sReserved;
    LONG    lMatch;
} FONTMETRICS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

cbMetrics Specifies the length (in bytes) of the font-metrics structure(s).

pcFonts Points to the variable that receives the number of fonts for which the application requires metrics.

pszFacename Points to the null-terminated string that specifies the face name.

flOptions Specifies whether to enumerate public or private fonts. This parameter may be any combination of the following values:

Value	Meaning
VQF_PUBLIC	Enumerate public fonts.
VQF_PRIVATE	Enumerate private fonts.

hvps Identifies the advanced video-input-and-output (AVIO) presentation space. This handle must have been created previously by using the **VioCreatePS** function.

Return Value The return value is the number of fonts not retrieved. The return value is -1 if an error occurs.

See Also **GpiQueryFonts, VioCreateLogFont, VioCreatePS**

■ VioQuerySetIds

```
USHORT VioQuerySetIds(palcids, pachNames, palTypes, cSets, hvps)
PLONG palcids;      /* pointer to array for local identifiers for fonts */
PSTR8 pachNames;    /* pointer to array for font names                  */
PLONG palTypes;     /* pointer to array for object types                */
LONG cSets;         /* number of local identifiers in use               */
HVPS hvps;          /* presentation-space handle                        */
```

The **VioQuerySetIds** function retrieves information about all available logical fonts. This function is similar to the **GpiQuerySetIds** function.

Parameters

palcids Points to the array that receives the local identifiers for the fonts.

pachNames Points to the array that receives the 8-character names for the fonts.

palTypes Points to the array that receives the object types for the fonts. All fonts have the object type LCIDT_FONT.

cSets Specifies the number of local identifiers currently in use and therefore the maximum number of objects for which information can be returned. You can determine this value by using the **GpiQueryNumberSetIds** function.

hvps Identifies the advanced video-input-and-output (AVIO) presentation space. This handle must have been created previously by using the **VioCreatePS** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

GpiQueryNumberSetIds, GpiQuerySetIds, VioCreatePS

■ VioReadCellStr

```
USHORT VioReadCellStr(pchCellString, pcb, usRow, usColumn, hvio)
PCH pchCellString;      /* pointer to buffer for string         */
PUSHORT pcb;            /* pointer to variable for string length */
USHORT usRow;           /* starting location (row)              */
USHORT usColumn;        /* starting location (column)           */
HVIO hvio;              /* video handle                         */
```

The **VioReadCellStr** function reads one or more cells (character-attribute pairs) from the screen, starting at the specified location. If the string is longer than the current line, the function continues reading it at the beginning of the next line but does not read past the end of the screen.

The **VioReadCellStr** function is a family API function.

Parameters

pchCellString Points to the buffer that receives the cell string.

pcb Points to the variable that specifies the length (in bytes) of the buffer. The length should be an even number. On return, this function copies the length of the string to the variable.

usRow Specifies the starting row of the cell string to read.

usColumn Specifies the starting column of the cell string to read.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioReadCellStr** to read Line 0, then calls the **VioWrtCellStr** function to write the cell string to Line 24:

```
CHAR achCells[160];
USHORT cb = sizeof(achCells);
VioReadCellStr(achCells,    /* buffer for string                      */
    &cb,                    /* pointer to variable for string length */
    0,                      /* starting location (row)                */
    0,                      /* starting location (column)             */
    0);                     /* video handle                           */
VioWrtCellStr(achCells, cb, 24, 0, 0);
```

See Also

VioReadCharStr, VioWrtCellStr

VioReadCharStr

```
USHORT VioReadCharStr(pchString, pcb, usRow, usColumn, hvio)
PCH pchString;          /* pointer to buffer for string             */
PUSHORT pcb;            /* pointer to variable for length of buffer */
USHORT usRow;           /* starting location (row)                  */
USHORT usColumn;        /* starting location (column)               */
HVIO hvio;              /* video handle                             */
```

The **VioReadCharStr** function reads a character string from the screen, starting at a specified location. If the character string is longer than the current line, the function continues reading it at the beginning of the next line but does not read past the end of the screen.

The **VioReadCharStr** function is a family API function.

Parameters

pchString Points to the buffer that receives the character string.

pcb Points to the variable that specifies the length (in bytes) of the buffer. On return, the function copies the length of the string to the variable.

usRow Specifies the starting row of the character to be read.

usColumn Specifies the starting column of the character to be read.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioReadCharStr** to read a character string that is 80 characters long, starting at Row 1, Column 0 of the screen. It then calls the **VioWrtCharStr** function to write the character string to Row 24, Column 0.

```
CHAR achString[80];
USHORT cb = sizeof(achString);
VioReadCharStr(achString,                /* string buffer    */
    &cb,                                 /* length of buffer */
    1,                                   /* row              */
    0,                                   /* column           */
    0);                                  /* video handle     */
VioWrtCharStr(achString, cb, 24, 0, 0);
```

See Also

VioReadCellStr, VioWrtCharStr

■ VioRegister

USHORT VioRegister(*pszModuleName, pszEntryName, flFunction1, flFunction2***)**

```
PSZ pszModuleName;     /* pointer to module name      */
PSZ pszEntryName;      /* pointer to entry-point name */
ULONG flFunction1;     /* function flag 1             */
ULONG flFunction2;     /* function flag 2             */
```

The **VioRegister** function registers a **Vio** subsystem within a screen group. **VioRegister** temporarily replaces one or more default **Vio** functions, as specified by the *flFunction1* and *flFunction2* parameters, with the functions pointed to by the *pszModuleName* parameter. Once **VioRegister** replaces a function, MS OS/2 passes any subsequent call to the replaced function to a function in the given module. If you do not replace a function, MS OS/2 continues to call the default **Vio** function.

Parameters

pszModuleName Points to the null-terminated string that specifies the name of the dynamic-link module containing the replacement **Vio** functions. The string must be a valid filename.

pszEntryName Points to the null-terminated string that specifies the dynamic-link entry-point name of the function that replaces the specified **Vio** functions. For a full description, see the following "Comments" section.

flFunction1 Specifies the **Vio** function(s) to replace. This parameter can be any combination of the following values:

Value	Meaning
VR_VIOGETCURPOS	Replace **VioGetCurPos**.
VR_VIOGETCURTYPE	Replace **VioGetCurType**.
VR_VIOGETMODE	Replace **VioGetMode**.
VR_VIOGETBUF	Replace **VioGetBuf**.
VR_VIOGETPHYSBUF	Replace **VioGetPhysBuf**.
VR_VIOSETCURPOS	Replace **VioSetCurPos**.
VR_VIOSETCURTYPE	Replace **VioSetCurType**.
VR_VIOSETMODE	Replace **VioSetMode**.
VR_VIOSHOWBUF	Replace **VioShowBuf**.

Value	Meaning
VR_VIOREADCHARSTR	Replace **VioReadCharStr**.
VR_VIOREADCELLSTR	Replace **VioReadCellStr**.
VR_VIOWRTNCHAR	Replace **VioWrtNChar**.
VR_VIOWRTNATTR	Replace **VioWrtNAttr**.
VR_VIOWRTNCELL	Replace **VioWrtNCell**.
VR_VIOWRTTTY	Replace **VioWrtTTY**.
VR_VIOWRTCHARSTR	Replace **VioWrtCharStr**.
VR_VIOWRTCHARSTRATT	Replace **VioWrtCharStrAtt**.
VR_VIOWRTCELLSTR	Replace **VioWrtCellStr**.
VR_VIOSCROLLUP	Replace **VioScrollUp**.
VR_VIOSCROLLDN	Replace **VioScrollDn**.
VR_VIOSCROLLLF	Replace **VioScrollLf**.
VR_VIOSCROLLRT	Replace **VioScrollRt**.
VR_VIOSETANSI	Replace **VioSetAnsi**.
VR_VIOGETANSI	Replace **VioGetAnsi**.
VR_VIOPRTSC	Replace **VioPrtSc**.
VR_VIOSCRLOCK	Replace **VioScrLock**.
VR_VIOSCRUNLOCK	Replace **VioScrUnLock**.
VR_VIOSAVREDRAWWAIT	Replace **VioSavRedrawWait**.
VR_VIOSAVREDRAWUNDO	Replace **VioSavRedrawUndo**.
VR_VIOPOPUP	Replace **VioPopUp**.
VR_VIOENDPOPUP	Replace **VioEndPopUp**.
VR_VIOPRTSCTOGGLE	Replace **VioPrtScToggle**.

flFunction2 Specifies the **Vio** function(s) to replace. This parameter can be any combination of the following values:

Value	Meaning
VR_VIOMODEWAIT	Replace **VioModeWait**.
VR_VIOMODEUNDO	Replace **VioModeUndo**.
VR_VIOGETFONT	Replace **VioGetFont**.
VR_VIOGETCONFIG	Replace **VioGetConfig**.
VR_VIOSETCP	Replace **VioSetCp**.
VR_VIOGETCP	Replace **VioGetCp**.
VR_VIOSETFONT	Replace **VioSetFont**.
VR_VIOGETSTATE	Replace **VioGetState**.
VR_VIOSETSTATE	Replace **VioSetState**.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_ASCIIZ
ERROR_VIO_INVALID_MASK
ERROR_VIO_REGISTER

Comments

MS OS/2 passes a **Vio** function to the given module by preparing the stack and calling the function pointed to by the *pszEntryName* parameter. The specified module must export the entry-point function name. The entry-point function must determine which function is being requested (by checking the function code on the stack), then pass control to the appropriate function in the module. The entry-point function may then access any additional parameters placed on the stack by the original call.

Only one process in a screen group may use the **VioRegister** function at any given time. That is, only one process at a time can replace **Vio** functions. The process can restore the default **Vio** functions by calling the **VioDeRegister** function. A process can replace **Vio** functions any number of times, but only by first restoring the default functions and then reregistering the new functions.

The entry-point function (*FuncName*) must have the following form:

SHORT FAR *FuncName*(*selDataSeg*, *usReserved1*, *fFunction*, *ulReserved2*, *usParam1*, *usParam2*, *usParam3*, *usParam4*, *usParam5*, *usParam6*)
SEL *selDataSeg*;
USHORT *usReserved1*;
USHORT *fFunction*;
ULONG *ulReserved2*;
USHORT *usParam1*;
USHORT *usParam2*;
USHORT *usParam3*;
USHORT *usParam4*;
USHORT *usParam5*;
USHORT *usParam6*;

Parameter	Description
selDataSeg	Specifies the data segment selector of the process calling the **Vio** function.
usReserved1	Specifies a reserved value that must not be changed. This value represents a return address for the MS OS/2 function that routes calls to **Vio** functions.
fFunction	Specifies the function code of the function request. This parameter can be one of the following values:

Value	Meaning
0x0000	**VioGetPhysBuf** called.
0x0001	**VioGetBuf** called.
0x0002	**VioShowBuf** called.
0x0003	**VioGetCurPos** called.

Value	Meaning
0x0004	**VioGetCurType** called.
0x0005	**VioGetMode** called.
0x0006	**VioSetCurPos** called.
0x0007	**VioSetCurType** called.
0x0008	**VioSetMode** called.
0x0009	**VioReadCharStr** called.
0x000A	**VioReadCellStr** called.
0x000B	**VioWrtNChar** called.
0x000C	**VioWrtNAttr** called.
0x000D	**VioWrtNCell** called.
0x000E	**VioWrtCharStr** called.
0x000F	**VioWrtCharStrAtt** called.
0x0010	**VioWrtCellStr** called.
0x0011	**VioWrtTTY** called.
0x0012	**VioScrollUp** called.
0x0013	**VioScrollDn** called.
0x0014	**VioScrollLf** called.
0x0015	**VioScrollRt** called.
0x0016	**VioSetAnsi** called.
0x0017	**VioGetAnsi** called.
0x0018	**VioPrtSc** called.
0x0019	**VioScrLock** called.
0x001A	**VioScrUnLock** called.
0x001B	**VioSavRedrawWait** called.
0x001C	**VioSavRedrawUndo** called.
0x001D	**VioPopUp** called.
0x001E	**VioEndPopUp** called.
0x001F	**VioPrtScToggle** called.
0x0020	**VioModeWait** called.
0x0021	**VioModeUndo** called.
0x0022	**VioGetFont** called.
0x0023	**VioGetConfig** called.
0x0024	**VioSetCp** called.
0x0025	**VioGetCp** called.
0x0026	**VioSetFont** called.
0x0027	**VioGetState** called.
0x0028	**VioSetState** called.

Parameter	Description
ulReserved2	Specifies a reserved value that must not be changed. This value represents the return address of the program that calls the specified **Vio** function.
usParam1-usParam6	Specifies up to six values passed with the original call to the **Vio** function. Not all requests include all six parameters since not all **Vio** functions use six parameters. The number and type of parameters used depend on the specific function.

The entry-point function should determine which function is requested and then carry out an appropriate action by using the passed parameters. The entry-point function can call a function within the same module to carry out the task. The entry-point or replacement function must leave the stack in the same state as it was received. This is required since the return addresses on the stack must be available in the correct order to return control to the program that originally called the **VioRegister** function.

The registered function should return -1 if it wants the original function called, 0 if no error occurred, or an error value.

In general, if the function needs to access the display, it must use the input-and-output control functions for the display. For more information, see Chapter 3, "Input-and-Output Control Functions."

The **VioRegister** function itself cannot be replaced.

If a process replaces the **VioPopUp** function, only the foreground process has access to the replacement function. Background processes continue to call the default **VioPopUp** function.

See Also

VioDeRegister, VioPopUp, VioSetCurPos

■ VioSavRedrawUndo

```
USHORT VioSavRedrawUndo(fRelinquish, fTerminate, hvio)
USHORT fRelinquish;   /* retain/relinquish ownership flag */
USHORT fTerminate;    /* terminate/continue flag          */
HVIO hvio;            /* video handle                     */
```

The **VioSavRedrawUndo** function cancels a request by a process to be notified when MS OS/2 switches screen groups. A process requests to be notified by calling the **VioSavRedrawWait** function. The request forces the calling thread to wait until a screen switch occurs. **VioSavRedrawUndo** cancels the request and allows the thread to continue (or terminates the thread, if requested to do so).

MS OS/2 permits only one process in a screen group to request screen switch notification. The first process to make a request owns it. Thereafter, other processes must wait for the owning process to relinquish the request before being given ownership. To force the process to relinquish ownership of the request, use the **VioSavRedrawUndo** function.

Only the process that owns the change-mode request can call the **VioSavRedrawUndo** function.

Parameters

fRelinquish Specifies whether a process should retain or relinquish ownership of the request. If this parameter is UNDOI_GETOWNER, the process relinquishes ownership and is canceled by this function. If the parameter is UNDOI_RELEASEOWNER, the process retains ownership and can repeat the request without competing with other processes.

fTerminate Specifies whether to terminate the thread waiting for the mode change. If this parameter is UNDOK_ERRORCODE, the thread continues and receives an error value from the **VioSavRedrawWait** function. If the parameter is UNDOK_TERMINATE, the thread terminates.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_FUNCTION_OWNED
ERROR_VIO_INVALID_PARMS
ERROR_VIO_NO_SAVE_RESTORE_THD

See Also

VioModeUndo, VioSavRedrawWait

■ VioSavRedrawWait

```
USHORT VioSavRedrawWait(fEvent, pfNotify, usReserved)
USHORT fEvent;        /* event flag                          */
PUSHORT pfNotify;     /* pointer to variable for notify flag */
USHORT usReserved;    /* must be zero                        */
```

The **VioSavRedrawWait** function waits for a screen switch to occur. When a switch occurs, MS OS/2 sets the variable pointed to by the *pfNotify* parameter to a value that indicates the type of changes. The thread may then save or restore the display depending on the value pointed to by the *pfNotify* parameter. The thread must also save or restore the complete video mode, the state information, the registers, and the contents of the physical video buffer.

MS OS/2 permits only one process in a screen group to wait for a screen switch. The first process to make a request owns it.

The **VioSavRedrawWait** function is used typically by graphics programs (or text-mode programs that change the video registers directly) to save and restore the screen before and after MS OS/2 switches from one screen group to another. Screen switching often changes the screen mode and video register values. A thread that calls the **VioSavRedrawWait** function waits until a screen switch occurs and is then given control so that it can save or restore the screen.

Parameters

fEvent Specifies the event flag of the event to wait for. If this flag is VSRWI_SAVEANDREDRAW, the function returns when the screen needs to be either saved or restored. If the flag is VSRWI_REDRAW, the function returns only when the screen needs to be restored.

pfNotify Points to the variable that receives the flag specifying the action to carry out in response to the given event. If this flag is VSWRN_SAVE, the thread saves the video buffer, the registers, and the state information. If the flag is VSWRN_REDRAW, the thread restores the video buffer, the registers, and the state information.

usReserved Specifies a reserved value. This parameter must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_FUNCTION_OWNED
ERROR_VIO_INVALID_PARMS
ERROR_VIO_RETURN

Comments

When an application is notified that it should save its screen image, it saves its physical video buffer, video mode, and any other information the application needs in order to redraw its screen.

The thread that calls **VioSavRedrawWait** should carry out all tasks directly related to saving and restoring the screen information. Whenever a screen switch occurs, the thread should save or restore the screen and call **VioSavRedrawWait** as quickly as possible. The thread can access the physical video buffer, if necessary, but since the thread may not be the foreground process, it must not use the **VioScrLock** function to lock the screen. The thread should not call MS OS/2 functions (neither directly nor indirectly through other functions) that may generate pop-up screens or error pop-up screens. Doing so may cause MS OS/2 to lock up (that is, each call of the thread generates a pop-up screen, which in turn calls the thread and generates another pop-up screen, and so on). You can use the **VioSavRedrawUndo** function to end the thread when it is no longer needed.

In some cases, a thread may receive a request to restore the screen before receiving a request to save the screen. For such requests, the thread must determine whether the given request is valid.

Programs that need to save and restore the screen after a pop-up screen should use the **VioModeWait** function.

See Also

VioGetPhysBuf, VioModeWait, VioSavRedrawUndo

VioScrLock

```
USHORT VioScrLock(fWait, pfNotLocked, hvio)
USHORT fWait;          /* wait/no-wait flag            */
PBYTE pfNotLocked;     /* pointer to variable for status */
HVIO hvio;             /* video handle                 */
```

The **VioScrLock** function locks the physical video buffer for a process. While the buffer is locked, no other process may lock it. This function is used typically to coordinate the output of graphics programs so that only one process writes to the physical video buffer at a time. The function indicates when the screen is locked by another process and is not available for writing, rather than denying processes access to the physical video buffer.

Only one process in a screen group may lock the screen. If the screen is already locked, **VioScrLock** either waits for the screen to become unlocked or returns immediately, as determined by the *fWait* parameter. Processes that lock the screen should unlock it by using the **VioScrUnLock** function as soon as they have completed the output.

If a screen-switch request occurs while the screen lock is in effect, the switch is held for at least thirty seconds. If the process does not unlock the screen before thirty seconds elapse, MS OS/2 suspends the process and switches the screen. The suspended process remains in the background until it is switched back to the foreground.

The **VioScrLock** function is a family API function.

Parameters

fWait Specifies the flag that determines whether the process is to wait until the screen input or output can occur. If this flag is LOCKIO_NOWAIT, the process returns immediately if the screen is not available. If the flag is LOCKIO_WAIT, the process waits for the screen to become available.

pfNotLocked Points to the variable that receives the flag specifying whether the screen is locked. If this flag is LOCK_SUCCESS, the screen is locked. If the flag is LOCK_FAIL, the screen is not locked.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_LOCK
ERROR_VIO_WAIT_FLAG

Restrictions

In real mode, the following restriction applies to the **VioScrLock** function:

- The function always indicates that the lock was successful.

Example

This example calls **VioScrLock** and waits until the screen lock can be performed (the process is in the foreground):

```
USHORT fNotLocked;
VioScrLock(LOCKIO_WAIT,     /* waits until I/O can take place  */
    &fNotLocked,            /* variable to receive lock status */
    0);                     /* video handle                    */
    .
    .
    .
VioScrUnLock(0);
```

See Also

VioGetPhysBuf, VioScrUnLock

VioScrollDn

USHORT VioScrollDn(*usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, *cbLines*, *pbCell*, *hvio* **)**

```
USHORT usTopRow;    /* top row                   */
USHORT usLeftCol;   /* left column               */
USHORT usBotRow;    /* bottom row                */
USHORT usRightCol;  /* right column              */
USHORT cbLines;     /* number of blank lines     */
PBYTE pbCell;       /* pointer to cell to write  */
HVIO hvio;          /* video handle              */
```

The **VioScrollDn** function scrolls the current screen downward.

The **VioScrollDn** function is a family API function.

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbLines Specifies the number of lines to be inserted at the top of the screen area being scrolled. If this parameter is zero, no lines are scrolled.

pbCell Points to a character/attribute pair, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbLines*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollDn** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbLines* to their maximum values. The function clears the screen by using the character/attribute pair pointed to by the *pbCell* parameter.

Example

This example creates a cell containing the space character (0x20) and a white character attribute (0x07 on an EGA color monitor), and calls **VioScrollDn** to clear the screen by using this cell. By changing the character attribute, you could change the background color of the screen while clearing it at the same time (using the value 0xFFFF for *usBotRow*, *usRightCol*, and *cbLines* clears the screen):

```
BYTE bCell[2];
bCell[0] = 0x20;          /* space character         */
bCell[1] = 0x07;          /* white attribute (EGA) */
VioScrollDn(0,            /* top row                 */
    0,                    /* left column             */
    0xFFFF,               /* bottom row              */
    0xFFFF,               /* right column            */
    0xFFFF,               /* number of lines         */
    bCell,                /* cell to write           */
    0);                   /* video handle            */
```

See Also

VioScrollLf, VioScrollRt, VioScrollUp

VioScrollLf

```
USHORT VioScrollLf(usTopRow, usLeftCol, usBotRow, usRightCol, cbColumns, pbCell, hvio)
USHORT usTopRow;       /* top row                    */
USHORT usLeftCol;      /* left column                */
USHORT usBotRow;       /* bottom row                 */
USHORT usRightCol;     /* right column               */
USHORT cbColumns;      /* number of blank columns    */
PBYTE pbCell;          /* pointer to the cell to write */
HVIO hvio;             /* video handle               */
```

The **VioScrollLf** function scrolls the current screen toward the left.

The **VioScrollLf** function is a family API function.

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbColumns Specifies the number of columns of spaces to be inserted at the right. If this parameter is zero, no columns are inserted.

pbCell Points to a character/attribute pair, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbColumns*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollLf** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbColumns* to their maximum values. The function clears the screen by using the character/attribute pair pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollLf** to fill the last ten columns at the right of the screen with red hearts on a black background (a value of 0xFFFF is used for *usBotRow* and *usRightCol*):

```
BYTE bCell[2];
bCell[0] = 0x03;        /* heart character       */
bCell[1] = 0x04;        /* red attribute (EGA) */
VioScrollLf(0,          /* top row               */
    0,                  /* left column           */
    0xFFFF,             /* bottom row            */
    0xFFFF,             /* right column          */
    10,                 /* columns               */
    bCell,              /* cell to write         */
    0);                 /* video handle          */
```

See Also **VioScrollDn, VioScrollRt, VioScrollUp**

VioScrollRt

USHORT VioScrollRt(*usTopRow***,** *usLeftCol***,** *usBotRow***,** *usRightCol***,** *cbColumns***,** *pbCell***,** *hvio***)**

```
USHORT usTopRow;     /* top row                   */
USHORT usLeftCol;    /* left column               */
USHORT usBotRow;     /* bottom row                */
USHORT usRightCol;   /* right column              */
USHORT cbColumns;    /* number of blank columns */
PBYTE pbCell;        /* pointer to cell to write  */
HVIO hvio;           /* video handle              */
```

The **VioScrollRt** function scrolls the current screen toward the right.

The **VioScrollRt** function is a family API function.

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbColumns Specifies the number of columns of spaces to be inserted at the left. If this parameter is zero, no columns are inserted.

pbCell Points to a character/attribute pair, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbColumns*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollUp** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow, usRightCol,* and *cbColumns* to their maximum values. The function clears the screen by using the character/attribute pair pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollRt** to fill the first ten columns at the left of the screen with red hearts on a black background (a value of 0xFFFF is used for *usBotRow* and *usRightCol*):

```
BYTE bCell[2];
bCell[0] = 0x03;         /* heart character      */
bCell[1] = 0x04;         /* red attribute (EGA) */
VioScrollRt(0,           /* top row              */
    0,                   /* left column          */
    0xFFFF,              /* bottom row           */
    0xFFFF,              /* right column         */
    10,                  /* columns              */
    bCell,               /* cell to write        */
    0);                  /* video handle         */
```

See Also

VioScrollDn, VioScrollLf, VioScrollUp

■ VioScrollUp

```
USHORT VioScrollUp(usTopRow, usLeftCol, usBotRow, usRightCol, cbLines, pbCell, hvio)
USHORT usTopRow;     /* top row                   */
USHORT usLeftCol;    /* left column               */
USHORT usBotRow;     /* bottom row                */
USHORT usRightCol;   /* right column              */
USHORT cbLines;      /* number of blank lines     */
PBYTE pbCell;        /* pointer to cell to write  */
HVIO hvio;           /* video handle              */
```

The **VioScrollUp** function scrolls the current screen upward.

The **VioScrollUp** function is a family API function.

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbLines Specifies the number of blank lines to insert at the bottom of the screen area being scrolled. If this parameter is zero, no lines are inserted.

pbCell Points to a character/attribute pair, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbLines*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollUp** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbLines* to their maximum values. The function clears the screen by using the character/attribute pair pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollUp** to scroll the entire screen up (by using the value 0xFFFF for *usBotRow*, *usRightCol*, and *cbLines*) and to fill the screen area left blank by the scrolling with spaces on a green background (0x22 on an EGA color monitor):

```
BYTE bCell[2];
bCell[0] = 0x20;            /* space character         */
bCell[1] = 0x22;            /* green attribute (EGA)   */
VioScrollUp(0,              /* top row                 */
    0,                      /* left column             */
    0xFFFF,                 /* bottom row              */
    0xFFFF,                 /* right column            */
    0xFFFF,                 /* number of lines         */
    bCell,                  /* cell to write           */
    0);                     /* video handle            */
VioSetCurPos(0, 0  0);
```

See Also

VioScrollDn, VioScrollLf, VioScrollRt

■ VioScrUnLock

USHORT VioScrUnLock(*hvio***)**
HVIO *hvio*;   /* video handle */

The **VioScrUnLock** function unlocks the screen previously locked by the process.

The **VioScrUnLock** function is a family API function.

Parameters

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_UNLOCK

Example

This example calls the **VioScrLock** function to lock the screen, then calls **VioScrUnLock** to unlock the screen:

```
USHORT fNotLocked;
VioScrLock(LOCKIO_WAIT, &fNotLocked, 0);
    .
    .
    .
VioScrUnLock(0);
```

See Also

VioScrLock

■ VioSetAnsi

USHORT VioSetAnsi(*fAnsi, hvio***)**
USHORT *fAnsi*; /* ANSI flag */
HVIO *hvio*; /* video handle */

The **VioSetAnsi** function enables or disables processing of ANSI escape sequences by setting or clearing the ANSI flag, which specifies whether the **VioWrtTTY** function processes ANSI escape sequences.

When a screen group is started, ANSI processing is enabled for the screen group.

Parameters

fAnsi Specifies the ANSI flag, which determines whether ANSI processing is enabled or disabled. If this flag is ANSI_ON, ANSI processing is enabled. If the flag is ANSI_OFF, ANSI processing is disabled.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example displays two identical strings. Before the first string is displayed, **VioSetAnsi** disables ANSI processing. As a result, the **VioWrtTTY** function displays the ANSI escape sequences as characters. Before **VioWrtTTY** displays the second string, **VioSetAnsi** enables ANSI processing, and the string is displayed in inverse video (black characters on a white background):

```
VioSetAnsi(ANSI_OFF, 0);          /* disables ANSI processing */
VioWrtTTY("\33[7mHello World\33[0m\n\r", 21, 0);
VioSetAnsi(ANSI_ON, 0);           /* enables ANSI processing  */
VioWrtTTY("\33[7mHello World\33[0m\n\r", 21, 0);
```

See Also

VioGetAnsi

■ VioSetCp

```
USHORT VioSetCp(usReserved, idCodePage, hvio)
USHORT usReserved;      /* must be zero       */
USHORT idCodePage;      /* code-page identifier */
HVIO hvio;              /* video handle       */
```

The **VioSetCp** function sets the code page for the current screen group. The code page defines the character set used to display characters on the screen.

Parameters

usReserved Specifies a reserved value; must be zero.

idCodePage Specifies the code-page identifier. This parameter can be any code-page identifier specified in the **codepage** command line in the *config.sys* file. If this parameter is 0x0000, the function uses the system default code page. The following are the valid code-page numbers:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_BAD_CP
ERROR_VIO_INVALID_HANDLE

Example

This example calls **VioSetCp** to set the current system code page to the standard United States code page:

```
if (VioSetCp(0,         /* must be zero         */
      437,              /* code-page identifier */
      0)) {             /* video handle         */
    VioWrtTTY("Code page not specified in CONFIG.SYS\n\r", 39, 0);
```

See Also

DosSetCp, VioGetCp

■ VioSetCurPos

```
USHORT VioSetCurPos(usRow, usColumn, hvio)
USHORT usRow;       /* row position    */
USHORT usColumn;    /* column position */
HVIO hvio;          /* video handle    */
```

The **VioSetCurPos** function sets the screen position of the cursor.

The **VioSetCurPos** function is a family API function.

Parameters

usRow Specifies the row position of the cursor, where zero is the top row.

usColumn Specifies the column position of the cursor, where zero is the left-most column.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioSetCurPos** to place the cursor in the first column of the last row on the screen, and then displays the text "Hello World!":

```
VioSetCurPos(24,                    /* cursor row    */
    0,                              /* cursor column */
    0);                             /* video handle  */
VioWrtTTY("Hello World!", 12, 0);
```

See Also

VioGetCurPos, VioSetCurType

■ VioSetCurType

USHORT VioSetCurType(*pvioci, hvio* **)**
PVIOCURSORINFO *pvioci*; /* pointer to structure for cursor characteristics */
HVIO *hvio*; /* video handle */

The **VioSetCurType** function sets the cursor type.

The cursor is a shared resource for all processes in a screen group. If one process changes it, it is changed for all processes in the group.

The **VioSetCurType** function is a family API function.

Parameters

pvioci Points to the **VIOCURSORINFO** structure that specifies the characteristics of the cursor. The **VIOCURSORINFO** structure has the following form:

```
typedef struct _VIOCURSORINFO {
    USHORT yStart;
    USHORT cEnd;
    USHORT cx;
    USHORT attr;
} VIOCURSORINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_WIDTH

Example This example calls **VioSetCurType** to set the current cursor type to a block cursor with 14 scan lines:

```
VIOCURSORINFO vioci;
vioci.yStart = 0;        /* beginning scan line for cursor */
vioci.cEnd = 13;         /* ending scan line, zero-based   */
vioci.cx = 0;            /* default width, one character   */
vioci.attr = 0;          /* normal attribute               */
VioSetCurType(&vioci, 0);
```

See Also **VioGetCurType, VioSetCurPos**

VioSetDeviceCellSize

USHORT VioSetDeviceCellSize(*cRows*, *cColumns*, *hvps***)**
SHORT *cRows*; /* cell height */
SHORT *cColumns*; /* cell width */
HVPS *hvps*; /* presentation-space handle */

The **VioSetDeviceCellSize** function sets the size of the device character cell.

Parameters

cRows Specifies the height (in pels) of the character cell.

cColumns Specifies the width (in pels) of the character cell.

hvps Identifies the advanced video-input-and-output (AVIO) presentation space. This handle must have been created previously by using the **VioCreatePS** function.

Return Value The return value is zero if the function is successful. Otherwise, it is an error value.

See Also **VioCreatePS, VioGetDeviceCellSize**

VioSetFont

USHORT VioSetFont(*pviofi*, *hvio***)**
PVIOFONTINFO *pviofi*; /* pointer to structure for display font */
HVIO *hvio*; /* video handle */

The **VioSetFont** function sets the font used to display characters on the screen. A font consists of several bitmaps, one for each character in a character set. The bitmaps define the character shapes. The font must be compatible with the current screen mode; that is, the bitmap size must match the current character-cell size.

The **VioSetFont** function resets the current code page. A subsequent call to the **VioGetCp** function returns an error value.

Not all display adapters permit the font to be set.

Parameters

pviofi Points to the **VIOFONTINFO** structure that specifies the display font. The **VIOFONTINFO** structure has the following form:

```
typedef struct _VIOFONTINFO {
    USHORT cb;
    USHORT type;
    USHORT cxCell;
    USHORT cyCell;
    ULONG  pbData;
    USHORT cbData;
} VIOFONTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_LENGTH

See Also

VioGetCp, **VioGetFont**

VioSetMode

USHORT VioSetMode(*pviomi*, *hvio***)**
PVIOMODEINFO *pviomi*; /* pointer to structure for screen mode */
HVIO *hvio*; /* video handle */

The **VioSetMode** function sets the screen mode. The screen mode defines the display mode (text or graphics), the number of colors being used (2, 4, or 16), and the width and height of the screen in both character cells and pels. **VioSetMode** also initializes the cursor position and type, but does not clear the screen.

The **VioSetMode** function is a family API function.

Parameters

pviomi Points to the **VIOMODEINFO** structure that specifies the screen mode. The **VIOMODEINFO** structure has the following form:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
} VIOMODEINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH
ERROR_VIO_MODE

Comments

Not all screen-mode values are valid for all displays.

Example

This example calls the **VioGetMode** function to retrieve the current display mode, changes the mode, and calls **VioSetMode** to enable the new display mode.

```
VIOMODEINFO viomi;
viomi.cb = sizeof(viomi);
VioGetMode(&viomi, 0);
if (viomi.vres > 350)                                  /* VGA display */
    viomi.row = (viomi.row == 50) ? 25 : 50;
else                                                   /* EGA display */
    viomi.row = (viomi.row == 43) ? 25 : 43;
VioSetMode(&viomi, 0);
```

See Also

VioGetMode, VioSetState

VioSetOrg

USHORT VioSetOrg(*sRow, sColumn, hvps***)**
SHORT *sRow*;   /* row number of cell */
SHORT *sColumn*;   /* column number of cell */
HVPS *hvps*;   /* presentation-space handle */

The **VioSetOrg** function sets the origin for an advanced video-input-and-output (AVIO) presentation space. It moves the specified character cell to the upper-left corner of the screen.

Parameters

sRow Specifies the row number of the character cell that is to be the origin.

sColumn Specifies the column number of the character cell that is to be the origin.

hvps Identifies the AVIO presentation space. This handle must have been created previously by using the **VioCreatePS** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

VioCreatePS, VioGetOrg

VioSetState

USHORT VioSetState(*pvoidState, hvio***)**
PVOID *pvoidState*;   /* pointer to buffer with new state */
HVIO *hvio*;   /* video handle */

The **VioSetState** function sets the palette-register values, the overscan (border) color, or the blink/background intensity switch.

Parameters

pvoidState Points to the structure that contains the request type and the values to set. The structure type, which depends on the request type specified in the **type** field of each structure, is one of the following: **VIOPALSTATE**, **VIOOVERSCAN**, or **VIOINTENSITY**. These structures have the following forms:

```
typedef struct _VIOPALSTATE {
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;

typedef struct _VIOOVERSCAN {
    USHORT cb;
    USHORT type;
    USHORT color;
} VIOOVERSCAN;

typedef struct _VIOINTENSITY {
    USHORT cb;
    USHORT type;
    USHORT fs;
} VIOINTENSITY;
```

Not all request-type values are valid for all screen modes.

For a full description, see Chapter 4, "Types, Macros, Structures."

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

Example

This example retrieves the current settings of the palette registers, switches palette registers #0 and #7, and calls **VioSetState** to enable the new settings:

```
BYTE abState[38];
PVIOPALSTATE pviopal;
USHORT usTmp;
pviopal = (PVIOPALSTATE) abState;
pviopal->cb = sizeof(abState);
pviopal->type = 0;              /* retrieves palette registers */
pviopal->iFirst = 0;            /* first register to retrieve  */
VioGetState(pviopal, 0);        /* retrieves current settings  */
usTmp = pviopal->acolor[0];     /* swaps# 0 and# 7             */
pviopal->acolor[0] = pviopal->acolor[7];
pviopal->acolor[7] = usTmp;
VioSetState(pviopal, 0);        /* enables new settings        */
```

See Also

VioGetState, VioSetMode

■ VioShowBuf

USHORT VioShowBuf(*offLVB*, *cbOutput*, *hvio* **)**

```
USHORT offLVB;      /* offset into logical video buffer */
USHORT cbOutput;    /* length                           */
HVIO hvio;          /* video handle                     */
```

The **VioShowBuf** function updates the physical screen from the logical video buffer (LVB). You may use the logical video buffer to directly manipulate information displayed on the screen.

The **VioShowBuf** function is a family API function.

Parameters

offLVB Specifies the offset into the logical video buffer at which the screen update is to start.

cbOutput Specifies the length (in bytes) of the screen area to update.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Example

This example retrieves the address of the logical video buffer, makes changes to that buffer, and calls **VioShowBuf** to update the physical video buffer from the logical video buffer:

```
PBYTE pbLVB;
USHORT cbOutput;
VioGetBuf((PULONG) &pbLVB, &cbOutput, 0);
    .
    .
    .
VioShowBuf(0,           /* offset into logical video buffer */
    cbOutput,           /* length of screen area            */
    0);                 /* video handle                     */
```

See Also

VioGetBuf, VioGetPhysBuf

■ VioShowPS

USHORT VioShowPS(*cRows*, *cColumns*, *off*, *hvps* **)**

```
SHORT cRows;        /* height of rectangle           */
SHORT cColumns;     /* width of rectangle            */
SHORT off;          /* upper-left corner of rectangle */
HVPS hvps;          /* presentation-space handle     */
```

The **VioShowPS** function updates the display by copying all the latest changes in the specified rectangle to the display.

Parameters

cRows Specifies the height (in character cells) of the rectangle to update.

cColumns Specifies the width (in character cells) of the rectangle to update.

off Specifies the position of the upper-left corner of the rectangle to update. The position is relative to the first character cell in the advanced video-input-and-output (AVIO) presentation space.

hvps Identifies the AVIO presentation space. This handle must have been created previously by using the **VioCreatePS** function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value.

See Also

VioCreatePS

VioWrtCellStr

```
USHORT VioWrtCellStr(pchCellString, cbCellString, usRow, usColumn, hvio)
PCH pchCellString;         /* pointer to cell string        */
USHORT cbCellString;       /* length of string              */
USHORT usRow;              /* starting position (row)       */
USHORT usColumn;           /* starting position (column)    */
HVIO hvio;                 /* video handle                  */
```

The **VioWrtCellStr** function writes a cell string to the screen. A cell string is one or more character/attribute pairs. A character/attribute pair defines the character to be written and the character attribute by which it is displayed.

If the string is longer than the current line, the function continues writing it at the beginning of the next line, but does not write past the end of the screen.

The **VioWrtCellStr** function is a family API function.

Parameters

pchCellString Points to the cell string to write.

cbCellString Specifies the length (in bytes) of the cell string. The length should be an even number.

usRow Specifies the row at which to start writing the cell string.

usColumn Specifies the column at which to start writing the cell string.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls the **VioWrtCellStr** function to display the string "Hello World!" using 12 different attributes:

```
CHAR achCellString[] = "H\1e\21\31\4o\5 \6W\7o\10r\111\13d\14!";
    .
    .
    .
VioWrtCellStr(achCellString,        /* character/attribute string */
    sizeof(achCellString),          /* length of string           */
    10,                             /* row                        */
    35,                             /* column                     */
    0);                             /* video handle               */
```

See Also

VioReadCellStr, VioWrtCharStr, VioWrtTTY

■ VioWrtCharStr

USHORT VioWrtCharStr(*pchString***,** *cbString***,** *usRow***,** *usColumn***,** *hvio***)**

```
PCH pchString;        /* pointer to string to write   */
USHORT cbString;      /* length of character string   */
USHORT usRow;         /* starting position (row)      */
USHORT usColumn;      /* starting position (column)   */
HVIO hvio;            /* video handle                 */
```

The **VioWrtCharStr** function writes a character string to the screen. A character string contains one or more character values, but no attributes. The function uses the present screen attributes to display the new characters. If the string is longer than the current line, the function continues writing it at the beginning of the next line but does not write past the end of the screen.

The **VioWrtCharStr** function is a family API function.

Parameters

pchString Points to the character string to write.

cbString Specifies the length (in bytes) of the character string.

usRow Specifies the row at which to start writing the string.

usColumn Specifies the column at which to start writing the string.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioWrtCharStr** to display the string "Hello World!" on the screen at Row 12, Column 30:

```
VioWrtCharStr("Hello World!",       /* string to display */
    12,                             /* length of string  */
    12,                             /* row               */
    30,                             /* column            */
    0);                             /* video handle      */
```

See Also

VioReadCharStr, VioWrtCharStr, VioWrtTTY

VioWrtCharStrAtt

USHORT VioWrtCharStrAtt(*pchString, cbString, usRow, usColumn, pbAttr, hvio***)**

```
PCH pchString;          /* pointer to string to write  */
USHORT cbString;        /* length of string            */
USHORT usRow;           /* starting position (row)     */
USHORT usColumn;        /* starting position (column)  */
PBYTE pbAttr;           /* pointer to attribute        */
HVIO hvio;              /* video handle                */
```

The **VioWrtCharStrAtt** function writes a character string to the screen, using the specified attribute. If the string is longer than the current line, the function continues writing it at the beginning of the next line but does not write past the end of the screen.

The **VioWrtCharStrAtt** function is a family API function.

Parameters

pchString Points to the character string to write.

cbString Specifies the length (in bytes) of the character string.

usRow Specifies the row at which to start writing the string.

usColumn Specifies the column at which to start writing the string.

pbAttr Points to the variable that specifies the attribute to be used for each character in the string.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioWrtCharStrAtt** to display the string "Hello World!" in the center of the screen in green characters on a white background (on an EGA color monitor):

```
BYTE bhAttr = 0x72;         /* green character, white background */
VioWrtCharStrAtt("Hello World!",        /* string to display    */
     12,                                /* length of string     */
     12,                                /* row                  */
     35,                                /* column               */
     &bhAttr,                           /* address of attribute */
     0);                                /* video handle         */
```

See Also

VioWrtCharStr, VioWrtNAttr, VioWrtTTY

■ VioWrtNAttr

```
USHORT VioWrtNAttr(pbAttr, cb, usRow, usColumn, hvio)
PBYTE pbAttr;        /* pointer to attribute to write */
USHORT cb;           /* number of times to write      */
USHORT usRow;        /* starting position (row)       */
USHORT usColumn;     /* starting position (column)    */
HVIO hvio;           /* video handle                  */
```

The **VioWrtNAttr** function writes a character attribute to the screen a specified number of times. If the attribute is repeated more times than can fit on the current line, the function continues writing it at the beginning of the next line but does not write past the end of the screen.

The **VioWrtNAttr** function is a family API function.

Parameters

pbAttr Points to the variable that specifies the character attribute to write.

cb Specifies the number of times to write the character attribute.

usRow Specifies the row at which to start writing the attribute.

usColumn Specifies the column at which to start writing the attribute.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioWrtNAttr** to change all the character attributes on the screen to green letters on a black background (on an EGA color monitor):

```
BYTE bAttr = 0x02;          /* green character, black background  */
VioWrtNAttr(&bAttr,         /* address of attribute               */
    25 * 80,                /* number of times to write attribute */
    0,                      /* row                                */
    0,                      /* column                             */
    0);                     /* video handle                       */
```

See Also

VioWrtCharStrAtt, VioWrtNCell

■ VioWrtNCell

```
USHORT VioWrtNCell(pbCell, cb, usRow, usColumn, hvio)
PBYTE pbCell;        /* pointer to cell to write      */
USHORT cb;           /* number of times to write      */
USHORT usRow;        /* starting position (row)       */
USHORT usColumn;     /* starting position (column)    */
HVIO hvio;           /* video handle                  */
```

The **VioWrtNCell** function writes a cell to the screen a specified number of times. A cell (also called a character/attribute pair) consists of two unsigned byte values that specify the character and attribute to be written.

If the number of times that a cell is repeated is greater than the screen width, the **VioWrtNCell** function continues writing the cell at the beginning of the next line but does not write past the end of the screen.

The **VioWrtNCell** function is a family API function.

Parameters

pbCell Points to the cell to write.

cb Specifies the number of times to write the cell.

usRow Specifies the row at which to start writing the cell.

usColumn Specifies the column at which to start writing the cell.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls the **VioWrtNCell** function to fill the screen with green capital letter A's (on an EGA color monitor):

```
BYTE abCell[2];           /* character/attribute pair */
abCell[0] = 'A';          /* character (letter A)     */
abCell[1] = 0x02;         /* attribute (green)        */
VioWrtNCell(abCell,       /* address of attribute     */
    80 * 25,              /* number of cells to write */
    0,                    /* row                      */
    0,                    /* column                   */
    0);                   /* video handle             */
```

See Also

VioWrtNChar

VioWrtNChar

```
USHORT VioWrtNChar(pchChar, cb, usRow, usColumn, hvio)
PCH pchChar;          /* pointer to character to write */
USHORT cb;            /* number of times to write      */
USHORT usRow;         /* starting position (row)       */
USHORT usColumn;      /* starting position (column)    */
HVIO hvio;            /* video handle                  */
```

The **VioWrtNChar** function writes a character to the screen a specified number of times. The function uses the present screen character attribute to display the new character.

If the character is repeated more times than can fit on the current line, the **VioWrtNChar** function continues writing it at the beginning of the next line but does not write past the end of the screen.

The **VioWrtNChar** function is a family API function.

Parameters

pchChar Points to the character to write.

cb Specifies the number of times to write the character.

usRow Specifies the row at which to start writing the character.

usColumn Specifies the column at which to start writing the character.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be one of the following:

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls the **VioWrtNChar** function to fill the screen with capital letter A's:

```
VioWrtNChar("A",        /* address of character          */
    80 * 25,            /* number of characters to write */
    0,                  /* row                           */
    0,                  /* column                        */
    0);                 /* video handle                  */
```

See Also

VioWrtNCell

VioWrtTTY

```
USHORT VioWrtTTY(pchString, cbString, hvio)
PCH pchString;       /* pointer to string to write */
USHORT cbString;     /* length of string           */
HVIO hvio;           /* video handle               */
```

The **VioWrtTTY** function writes a character string to the screen, starting at the current cursor position. This function advances the cursor as it writes each character, using a default attribute for each character. If the function reaches the end of the line, it continues writing at the beginning of the next line. If it reaches the end of the last line on the screen, it scrolls the screen and continues writing at the beginning of a new line.

The **VioWrtTTY** function is a family API function.

Parameters

pchString Points to the character string to write.

cbString Specifies the length (in bytes) of the character string.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value, which may be the following:

ERROR_VIO_INVALID_HANDLE

Comments

For some ASCII values, **VioWrtTTY** carries out an action rather than displaying a character. The following list describes the action taken when the given ASCII byte value is in the string:

Value	Meaning
0x08	BACKSPACE. Move the cursor left by one position, without deleting any character that is under the cursor. If the cursor is at the beginning of the line, take no action.
0x09	TAB. Copy spaces from the current cursor position to the next tab stop. Tab stops are placed at every eighth character position on a line.
0x0A	LINEFEED. Move the cursor down to the next line. The screen will scroll up one line if the current line is at the bottom of the screen.
0x0D	RETURN. Move the cursor to the beginning of the line.
0x07	Bell. Generate a beep on the computer's speaker.

If the process has enabled ANSI processing by using the **VioSetAnsi** function, **VioWrtTTY** processes any ANSI escape sequences in the string.

Example

The following example calls **VioWrtTTY** to write a message to the screen and beep the computer's speaker:

```
VioWrtTTY("File not found\r\n\007", 17, 0);
```

See Also

VioSetCurPos, VioWrtCellStr, VioWrtCharStr

Chapter 3

Input-and-Output Control Functions

3.1 Introduction

This chapter describes the input-and-output control (**IOCtl**) functions. A program can send commands to and retrieve data from a device driver by using the **DosDevIOCtl** function. The **DosDevIOCtl** function sends the specified codes and data directly to the given device driver, which then carries out the specified action.

IOCtl functions typically are used to get information about or data from a device driver that is not available through standard MS OS/2 functions. For example, **IOCtl** functions can be used to set the baud rate of a serial port or read input from a mouse.

3.2 Category and Function Codes

Each **IOCtl** function has a category and a function code. The category code defines the type of device to be accessed. MS OS/2 has several predefined categories. In general, all codes in the range 0x0000 through 0x007F are reserved for predefined categories. A device driver may also use additional categories, but these must be explicitly defined by the device and be in the range 0x0080 through 0x00FF. The following list shows which devices correspond to the given categories:

Category	Device
0x0001	Serial-device control
0x0003	Screen/pointer-draw control
0x0004	Keyboard control
0x0005	Printer control
0x0006	Light-pen control (Reserved)
0x0007	Pointing-device (mouse) control
0x0008	Disk/diskette control
0x0009	Physical-disk control
0x000A	Character-monitor control
0x000B	General device control

The function code defines the action to carry out, such as reading from or writing to the device and retrieving or setting the device modes. The number and meaning of each function code depend on the device driver and the specified category. Function codes range from 0x0000 through 0x001F and are combined with one or more of the following values:

Value	Meaning
0x0020	Retrieve data or information from the device. If 0x0020 is not part of the code, the function sends data or commands to the device.

Value	Meaning
0x0040	Pass the command to the device driver. If 0x0040 is not part of the code, MS OS/2 intercepts the command.
0x0080	Ignore the command if the device driver does not support it. If 0x0080 is not part of the code, the function returns an error code if the command is not supported.

The following table lists the **IOCtl** functions by category and function codes and shows the corresponding function name:

Table 3.1 Specific Category and Function Codes

Serial-Device Control

Category, Function	Function name
0x0001,0x0041	ASYNC_SETBAUDRATE
0x0001,0x0042	ASYNC_SETLINECTRL
0x0001,0x0044	ASYNC_TRANSMITIMM
0x0001,0x0045	ASYNC_SETBREAKOFF
0x0001,0x0046	ASYNC_SETMODEMCTRL
0x0001,0x004B	ASYNC_SETBREAKON
0x0001,0x0047	ASYNC_STOPTRANSMIT
0x0001,0x0048	ASYNC_STARTTRANSMIT
0x0001,0x0053	ASYNC_SETDCBINFO
0x0001,0x0061	ASYNC_GETBAUDRATE
0x0001,0x0062	ASYNC_GETLINECTRL
0x0001,0x0064	ASYNC_GETCOMMSTATUS
0x0001,0x0065	ASYNC_GETLINESTATUS
0x0001,0x0066	ASYNC_GETMODEMOUTPUT
0x0001,0x0067	ASYNC_GETMODEMINPUT
0x0001,0x0068	ASYNC_GETINQUECOUNT
0x0001,0x0069	ASYNC_GETOUTQUECOUNT
0x0001,0x006D	ASYNC_GETCOMMERROR
0x0001,0x0072	ASYNC_GETCOMMEVENT
0x0001,0x0073	ASYNC_GETDCBINFO

Screen/Pointer-Draw Control

Category, Function	Function name
0x0003, 0x0072	PTR_GETPTRDRAWADDRESS

Keyboard Control

Category, Function	Function name
0x0004,0x0050	KBD_SETTRANSTABLE
0x0004,0x0051	KBD_SETINPUTMODE

Table 3.1 *(Continued)*

Category, Function	Function name
0x0004,0x0052	KBD_SETINTERIMFLAG
0x0004,0x0053	KBD_SETSHIFTSTATE
0x0004,0x0054	KBD_SETTYPAMATICRATE
0x0004,0x0055	KBD_SETFGNDSCREENGRP
0x0004,0x0056	KBD_SETSESMGRHOTKEY
0x0004,0x0057	KBD_SETFOCUS
0x0004,0x0058	KBD_SETKCB
0x0004,0x005C	KBD_SETNLS
0x0004,0x005D	KBD_CREATE
0x0004,0x005E	KBD_DESTROY
0x0004,0x0071	KBD_GETINPUTMODE
0x0004,0x0072	KBD_GETINTERIMFLAG
0x0004,0x0073	KBD_GETSHIFTSTATE
0x0004,0x0074	KBD_READCHAR
0x0004,0x0075	KBD_PEEKCHAR
0x0004,0x0076	KBD_GETSESMGRHOTKEY
0x0004,0x0077	KBD_GETKEYBDTYPE
0x0004,0x0078	KBD_GETCODEPAGEID
0x0004,0x0079	KBD_XLATESCAN

Printer Control

Category, Function	Function name
0x0005,0x0042	PRT_SETFRAMECTL
0x0005,0x0044	PRT_SETINFINITERETRY
0x0005,0x0046	PRT_INITPRINTER
0x005,0x0048	PRT_ACTIVATEFONT
0x0005,0x0062	PRT_GETFRAMECTL
0x0005,0x0064	PRT_GETINFINITERETRY
0x0005,0x0066	PRT_GETPRINTERSTATUS
0x0005,0x0069	PRT_QUTRYACTIVEFONT
0x0005,0x006A	PRT_VERIFYFONT

Pointing-Device (Mouse) Control

Category, Function	Function name
0x0007,0x0050	MOU_ALLOWPTRDRAW
0x0007,0x0051	MOU_UPDATEDISPLAYMODE
0x0007,0x0052	MOU_SCREENSWITCH
0x0007,0x0053	MOU_SETSCALEFACTORS
0x0007,0x0054	MOU_SETEVENTMASK

Table 3.1 *(Continued)*

Category, Function	Function name
0x0007,0x0055	MOU_SETHOTKEYBUTTON
0x0007,0x0056	MOU_SETPTRSHAPE
0x0007,0x0057	MOU_DRAWPTR
0x0007,0x0058	MOU_REMOVEPTR
0x0007,0x0059	MOU_SETPTRPOS
0x0007,0x005A	MOU_SETPROTDRAWADDRESS
0x0007,0x005B	MOU_SETREALDRAWADDRESS
0x0007,0x005C	MOU_SETMOUSTATUS
0x0007,0x0060	MOU_GETBUTTONCOUNT
0x0007,0x0061	MOU_GETMICKEYCOUNT
0x0007,0x0062	MOU_GETMOUSTATUS
0x0007,0x0063	MOU_READEVENTQUE
0x0007,0x0064	MOU_GETQUESTATUS
0x0007,0x0065	MOU_GETEVENTMASK
0x0007,0x0066	MOU_GETSCALEFACTORS
0x0007,0x0067	MOU_GETPTRPOS
0x0007,0x0068	MOU_GETPTRSHAPE
0x0007,0x0069	MOU_GETHOTKEYBUTTON

Disk/Diskette Control

Category, Function	Function name
0x0008,0x0000	DSK_LOCKDRIVE
0x0008,0x0001	DSK_UNLOCKDRIVE
0x0008,0x0002	DSK_REDETERMINEMEDIA
0x0008,0x0003	DSK_SETLOGICALMAP
0x0008,0x0020	DSK_BLOCKREMOVABLE
0x0008,0x0021	DSK_GETLOGICALMAP
0x0008,0x0043	DSK_SETDEVICEPARAMS
0x0008,0x0044	DSK_WRITETRACK
0x0008,0x0045	DSK_FORMATVERIFY
0x0008,0x0063	DSK_GETDEVICEPARAMS
0x0008,0x0064	DSK_READTRACK
0x0008,0x0065	DSK_VERIFYTRACK

Physical-Disk Control

Category, Function	Function name
0x0009,0x0000	PDSK_LOCKPHYSDRIVE
0x0009,0x0001	PDSK_UNLOCKPHYSDRIVE

Table 3.1 *(Continued)*

Category, Function	Function name
0x0009,0x0044	PDSK_WRITEPHYSTRACK
0x0009,0x0063	PDSK_GETPHYSDEVICEPARAMS
0x0009,0x0064	PDSK_READPHYSTRACK
0x0009,0x0065	PDSK_VERIFYPHYSTRACK

Character-Monitor Control

Category, Function	Function name
0x000A,0x0040	MON_REGISTERMONITOR

General Device Control

Category, Function	Function name
0x000B,0x0001	DEV_FLUSHINPUT
0x000B,0x0002	DEV_FLUSHOUTPUT
0x000B,0x0060	DEV_QUERYMONSUPPORT

3.3 Functions

This section lists the **IOCtl** functions in alphabetical order. Each function's syntax is given and the parameters and return values are described.

ASYNC_GETBAUDRATE

```
USHORT DosDevIOCtl(pusBaudRate, 0L, 0x0061, 0x0001, hDevice)
PUSHORT pusBaudRate;    /* pointer to variable for baud rate */
HFILE hDevice;          /* device handle                     */
```

The ASYNC_GETBAUDRATE function retrieves the baud rate for the specified serial device. The baud rate specifies the number of bits per second that the serial device transmits or receives.

Parameters

pusBaudRate Points to the variable that receives the baud rate.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, ASYNC_SETBAUDRATE

ASYNC_GETCOMMERROR

```
USHORT DosDevIOCtl(pfCommErr, 0L, 0x006D, 0x0001, hDevice)
PUSHORT pfCommErr;    /* pointer to variable for error */
HFILE hDevice;        /* device handle                 */
```

The ASYNC_GETCOMMERROR function retrieves the communication error word. After copying the error-word value to the specified variable, the function clears the error word.

Parameters

pfCommErr Points to the variable that receives the communication status of the device. This variable can be a combination of the following values:

Value	Meaning
RX_QUE_OVERRUN	Receive-queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.
RX_HARDWARE_OVERRUN	Receive-hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.
PARITY_ERROR	The hardware detected a parity error.
FRAMING_ERROR	The hardware detected a framing error.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful. When an error occurs, the function returns an error value, and any value copied to the variable pointed to by the *pfCommErr* parameter is not valid, and the function does not clear the error word.

Comments

Other than using this function, the only way to clear the communications error word for a device is to open the device when there are no outstanding open handles for it. For more information, see the ASYNC_SETDCBINFO function (0x0001, 0x0053).

See Also

DosOpen, ASYNC_GETCOMMEVENT, ASYNC_GETCOMMSTATUS, ASYNC_SETDCBINFO

ASYNC_GETCOMMEVENT

```
USHORT DosDevIOCtl(pfEvent, 0L, 0x0072, 0x0001, hDevice)
PUSHORT pfEvent;    /* pointer to variable for events */
HFILE hDevice;      /* device handle                  */
```

The ASYNC_GETCOMMEVENT function retrieves the communications event flags from the internally maintained event word. After the function copies the event flags to the specified variable, it clears the event word.

Parameters

pfEvent Points to the variable that receives the event flags. This variable can be a combination of the following values:

Value	Meaning
CHAR_RECEIVED	A character has been read from the serial-device receive hardware and placed in the receive queue.
LAST_CHAR_SENT	The last character in the device-driver transmit queue has been sent to the serial-device transmit hardware. This does not mean there is no data to send in any outstanding write requests.
CTS_CHANGED	The clear-to-send (CTS) signal has changed state.
DSR_CHANGED	The data-set-ready (DSR) signal has changed state.
DCD_CHANGED	The data-carrier-detect (DCD) signal has changed state.
BREAK_DETECTED	A break has been detected.
ERROR_OCCURRED	A parity, framing, or overrun error has occurred. An overrun can be a receive hardware overrun or a receive queue overrun.
RI_DETECTED	The trailing edge of the ring indicator (RI) has been detected.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

This function clears the event word only when it is successful. The event word remains unchanged until the device is fully closed (there are no outstanding open handles) and then reopened.

See Also

DosOpen, ASYNC_GETCOMMSTATUS, ASYNC_GETCOMMERROR

ASYNC_GETCOMMSTATUS

```
USHORT DosDevIOCtl(pbStatus, 0L, 0x0064, 0x0001, hDevice)
PBYTE pbStatus;    /* pointer to variable for status */
HFILE hDevice;     /* device handle                  */
```

The ASYNC_GETCOMMSTATUS function retrieves the communication status of the specified device.

Parameters

pbStatus Points to the variable that receives the communication status. This variable can be a combination of the following values:

Value	Meaning
TX_WAITING_FOR_CTS	Transmission is waiting for the clear-to-send (CTS) signal to be turned on. For a full description, see the ASYNC_SETDCBINFO function (0x0001, 0x0053).
TX_WAITING_FOR_DSR	Transmission is waiting for the data-set-ready (DSR) signal to be turned on. For a full description, see the ASYNC_SETDCBINFO function (0x0001, 0x0053).
TX_WAITING_FOR_DCD	Transmission is waiting for the data-carrier-detected (DCD) signal to be turned on. For a full description, see the ASYNC_SETDCBINFO function (0x0001, 0x0053).
TX_WAITING_FOR_XON	Transmission is waiting because the XOFF character is received. For a full description, see the following "Comments" section.
TX_WAITING_TO_SEND_XON	Transmission is waiting because the XOFF character is transmitted. For a full description, see the following "Comments" section.
TX_WAITING_WHILE_BREAK_ON	Transmission is waiting because a break is being transmitted. For a full description, see the ASYNC_SETBREAKON function (0x0001, 0x004B).
TX_WAITING_TO_SEND_IMM	Character is waiting to transmit immediately. For a full description, see the ASYNC_TRANSMITIMM function (0x0001, 0x0044).
RX_WAITING_FOR_DSR	Receive state is waiting for the data-set-ready (DSR) signal to be turned on. For a full description, see the ASYNC_SETDCBINFO function (0x0001, 0x0053).

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments Transmit status indicates why transmission is not occurring, regardless of whether or not there is data to transmit. However, the device driver must be enabled for the given condition (for example, enabled for output handshaking for the modem-control signal) for the status to reflect that the device driver is waiting for the given condition to transmit.

For example, TX_WAITING_FOR_CTS means that the device driver puts receive characters in the device-driver receive queue, the device driver is not waiting to transmit a character immediately, and characters from the device-driver transmit queue are not transmitted because the clear-to-send (CTS) signal for output handshaking is used and CTS does not have the proper value.

The communication status can include TX_WAITING_TO_SEND_XON if the device driver is enabled for automatic transmit flow control (XON/XOFF) or if the ASYNC_STOPTRANSMIT function (0x0001, 0x0047) has been used to tell the device driver to function as if an XOFF character is received. The ASYNC_TRANSMITIMM function (0x0001, 0x0044) can still be used to transmit characters immediately. The device driver can still automatically transmit XON and XOFF characters due to automatic receive flow control (XON/XOFF) when the device driver is in this state.

The communication status can include TX_WAITING_FOR_XON if the device driver is enabled for automatic receive flow control. When in this state, the ASYNC_TRANSMITIMM function (0x0001, 0x0044) can still be used to transmit characters immediately, and the device driver can still automatically transmit XON characters.

See Also **DosOpen**, ASYNC_GETCOMMEVENT, ASYNC_GETLINESTATUS, ASYNC_SETDCBINFO, ASYNC_STARTTRANSMIT, ASYNC_STOPTRANSMIT, ASYNC_TRANSMITTIMM

■ ASYNC_GETDCBINFO

USHORT DosDevIOCtl(*pusDCB*, **0L, 0x0073, 0x0001,** *hDevice***)**

```
PUSHORT pusDCB;    /* pointer to structure for device-control information */
HFILE hDevice;     /* device handle                                       */
```

The ASYNC_GETDCBINFO function retrieves device-control block information.

Parameters *pusDCB* Points to the **DCBINFO** structure that receives the device-control block information. The **DCBINFO** structure has the following form:

```
typedef struct _DCBINFO {
    USHORT usWriteTimeout;
    USHORT usReadTimeout;
    BYTE   bFlags1
    BYTE   bFlags2
    BYTE   bFlags3
    BYTE   bErrorReplacementChar;
    BYTE   bBreakReplacementChar;
    BYTE   bXONChar;
    BYTE   bXOFFChar;
} DCBINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful. When an error occurs, the function returns an error value, and any data copied to the **DCBINFO** structure pointed to by the *pusDCB* parameter is not valid.

Comments To ensure that only valid values are set in the device-control block, the program should call the ASYNC_GETDCBINFO function to fill the block, and then modify the settings and call the ASYNC_SETDCBINFO function with the modified block.

See Also **DosOpen**, ASYNC_SETDCBINFO

■ ASYNC_GETINQUECOUNT

USHORT DosDevIOCtl(*pcReceiveQue***, 0L, 0x0068, 0x0001,** *hDevice***)**
PUSHORT *pcReceiveQue***;** /* pointer to structure for character count */
HFILE *hDevice***;** /* device handle */

The ASYNC_GETINQUECOUNT function retrieves the number of characters in the receive queue.

Parameters *pcReceiveQue* Points to the **RXQUEUE** structure that receives the count of characters in the receive queue. The **RXQUEUE** structure has the following form:

```
typedef struct _RXQUEUE {
    USHORT cbChars;
    USHORT cbQueue;
} RXQUEUE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The device-driver receive queue is a memory buffer between the memory pointed to by the read-request packet and the receive hardware for this serial device. The application may not assume that there are no unsatisfied read requests if there are characters in the device-driver receive queue. The behavior of data movement between the read request and the receive queue may change from release to release of the device driver. Programs should not be written to have a dependency on this information.

Programs should be written to be independent of the receive queue being a fixed size. The information in this field allows the application to get the size of the receive queue. The current size of the receive queue is approximately 1K but is subject to change.

The application should be written to avoid device-driver receive queue overruns by using an application-to-application block protocol with the system the application is communicating with.

See Also **DosOpen**, ASYNC_GETOUTQUECOUNT

ASYNC_GETLINECTRL

USHORT DosDevIOCtl(*pbLineCtrl*, **0L, 0x0062, 0x0001,** *hDevice***)**
PBYTE *pbLineCtrl*; /* pointer to structure for control settings */
HFILE *hDevice*; /* device handle */

The ASYNC_GETLINECTRL function retrieves the line characteristics (stop bits, parity, data bits, break) for the specified device.

Parameters

pbLineCtrl Points to a **LINECONTROL** structure that receives the settings for the number of data bits, parity, and number of stop bits. The **LINECONTROL** structure has the following form:

```
typedef struct _LINECONTROL {
    BYTE bDataBits;
    BYTE bParity;
    BYTE bStopBits;
    BYTE fbTransBreak;
} LINECONTROL;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, ASYNC_SETLINECTRL

ASYNC_GETLINESTATUS

USHORT DosDevIOCtl(*pbTransStatus*, **0L, 0x0065, 0x0001,** *hDevice***)**
PBYTE *pbTransStatus*; /* pointer to variable for status */
HFILE *hDevice*; /* device handle */

The ASYNC_GETLINESTATUS function retrieves the data-transmission status for the specified serial device.

Parameters

pbTransStatus Points to the variable that receives the data-transmission status. This variable can be a combination of the following values:

Value	Meaning
WRITE_REQUEST_QUEUED	Write-request packets in progress or queued.
DATA_IN_TX_QUE	Data in the device-driver transmit queue.
HARDWARE_TRANSMITTING	Transmit hardware currently transmitting data.
CHAR_READY_TO_SEND_IMM	Character waiting to be transmitted immediately.

Value	Meaning
WAITING_TO_SEND_XON	Waiting to automatically transmit XON.
WAITING_TO_SEND_XOFF	Waiting to automatically transmit XOFF.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, ASYNC_GETCOMMSTATUS

■ ASYNC_GETMODEMINPUT

```
USHORT DosDevIOCtl(pbCtrlSignals, 0L, 0x0067, 0x0001, hDevice)
PBYTE pbCtrlSignals;    /* pointer to variable for control signals */
HFILE hDevice;          /* device handle                           */
```

The ASYNC_GETMODEMINPUT function retrieves the modem-control input signals for the specified device.

Parameters *pbCtrlSignals* Points to the variable that receives the modem-control signals. This variable can be a combination of the following values:

Value	Meaning
CTS_ON	Clear-to-send (CTS) signal is on. If not given, the signal is off.
DSR_ON	Data-set-ready (DSR) signal is on. If not given, the signal is off.
RI_ON	Ring-indicator (RI) signal is on. If not given, the signal is off.
DCD_ON	Data-carrier-detect (DCD) signal is on. If not given, the signal is off.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, ASYNC_GETMODEMOUTPUT, ASYNC_SETMODEMCTRL

■ ASYNC_GETMODEMOUTPUT

```
USHORT DosDevIOCtl(pbCtrlSignals, 0L, 0x0066, 0x0001, hDevice)
PBYTE pbCtrlSignals;    /* pointer to variable for control signals */
HFILE hDevice;          /* device handle                           */
```

The ASYNC_GETMODEMOUTPUT function retrieves the modem-control output signals for the specified device.

Parameters

pbCtrlSignals Points to the variable that receives the modem-control signals. This variable can be one or both of the following values:

Value	Meaning
DTR_ON	Data-terminal-ready (DTR) signal is on. If not given, the signal is off.
RTS_ON	Request-to-send (RTS) signal is on. If not given, the signal is off.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, ASYNC_GETMODEMINPUT, ASYNC_SETMODEMCTRL

ASYNC_GETOUTQUECOUNT

```
USHORT DosDevIOCtl(pcTransmitQue, 0L, 0x0069, 0x0001, hDevice)
PUSHORT pcTransmitQue;    /* pointer to structure for character count */
HFILE hDevice;            /* device handle                            */
```

The ASYNC_GETOUTQUECOUNT function retrieves a count of characters in the transmit queue.

Parameters

pcTransmitQue Points to the **RXQUEUE** structure that receives the count of characters in the transmit queue. The **RXQUEUE** structure has the following form:

```
typedef struct _RXQUEUE {
    USHORT cbChars;
    USHORT cbQueue;
} RXQUEUE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

The device-driver transmit queue is a memory buffer between the memory pointed to by the write-request packet and the transmit hardware for this serial device. If the transmit queue is empty, the program may not assume that all write requests are completed or that no write requests are outstanding. The behavior of data movement between the write request and the transmit queue may change from release to release of the device driver. Programs should not be written to have a dependency on this information.

Programs should be written to be independent of the transmit queue being a fixed size. The information in this field allows the application to get the size of the transmit queue. The current size of the transmit queue is approximately 128 bytes but is subject to change.

See Also

DosOpen, ASYNC_GETINQUECOUNT

■ ASYNC_SETBAUDRATE

```
USHORT DosDevIOCtl(0L, pusBitRate, 0x0041, 0x0001, hDevice)
PUSHORT pusBitRate;   /* pointer to variable with baud rate */
HFILE hDevice;        /* device handle                      */
```

The ASYNC_SETBAUDRATE function sets the baud rate for the specified serial device. The baud rate specifies the number of bits per second that the serial device transmits or receives.

Parameters

pusBitRate Points to the variable that contains the baud rate. This parameter can be any one of the following values: 110, 150, 300, 600, 1200, 2400, 4800, 9600, or 19200.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if the specified baud rate is out of range or an error occurs.

Comments

The initial rate for a serial device is 1200 baud. Once the rate is set, it remains unchanged until set again, even if the device is closed and then reopened.

See Also

DosOpen, ASYNC_GETBAUDRATE

■ ASYNC_SETBREAKOFF

```
USHORT DosDevIOCtl(pfCommErr, 0L, 0x0045, 0x0001, hDevice)
PUSHORT pfCommErr;    /* pointer to variable for error value */
HFILE hDevice;        /* device handle                       */
```

The ASYNC_SETBREAKOFF function turns off the break character. The device driver stops generating a break signal. It is not considered an error if the device driver is not generating a break signal. The device driver then resumes transmitting characters, taking into account all the other reasons why it may or may not transmit characters.

Parameters

pfCommErr Points to the variable that receives the communication status of the device. This variable can be a combination of the following values:

Value	Meaning
RX_QUE_OVERRUN	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.
RX_HARDWARE_OVERRUN	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.
PARITY_ERROR	The hardware detected a parity error.
FRAMING_ERROR	The hardware detected a framing error.

The function sets the variable to zero if it encounters an error.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, ASYNC_SETBREAKON

■ ASYNC_SETBREAKON

```
USHORT DosDevIOCtl(pfCommErr, 0L, 0x004B, 0x0001, hDevice)
PUSHORT pfCommErr;    /* pointer to variable for error value */
HFILE hDevice;        /* device handle                       */
```

The ASYNC_SETBREAKON function turns on the break character. The device driver generates the break signal immediately. It is not considered an error if the device driver is already generating a break signal. The device driver does not wait for the transmit hardware to become empty. However, more data will not be given to the transmit hardware until the break is turned off. The break signal will always be transmitted, regardless of whether the device driver is or is not transmitting characters due to other reasons.

Parameters *pfCommErr* Points to the variable that receives the communication status of the device. This variable can be a combination of the following values:

Value	Meaning
RX_QUE_OVERRUN	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.
RX_HARDWARE_OVERRUN	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.
PARITY_ERROR	The hardware detected a parity error.
FRAMING_ERROR	The hardware detected a framing error.

The function sets the variable to zero if it encounters an error.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments Closing the device turns off the break character if there are no outstanding open device handles.

See Also **DosOpen**, ASYNC_SETBREAKOFF

■ ASYNC_SETDCBINFO

USHORT DosDevIOCtl(0L, *pusDCB*, **0x0053, 0x0001,** *hDevice***)**
PUSHORT *pusDCB*; /* pointer to structure with device-control information */
HFILE *hDevice*; /* device handle */

The ASYNC_SETDCBINFO function sets device-control block information.

Parameters

pusDCB Points to the **DCBINFO** structure that receives the device-control block information. The **DCBINFO** structure has the following form:

```
typedef struct _DCBINFO {
    USHORT usWriteTimeout;
    USHORT usReadTimeout;
    BYTE   bFlags1
    BYTE   bFlags2
    BYTE   bFlags3
    BYTE   bErrorReplacementChar;
    BYTE   bBreakReplacementChar;
    BYTE   bXONChar;
    BYTE   bXOFFChar;
} DCBINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful. When an error occurs, the function returns an error value, and the device-control block characteristics of the device driver for this serial device remain unchanged.

Comments

A program can prevent making unwanted changes to device modes by calling the ASYNC_GETDCBINFO function (0x0001,0x0073) to retrieve a copy of the current DCB. The program can then modify only those fields it needs to and use the modified DCB with the ASYNC_SETDCBINFO function.

See Also

DosOpen, ASYNC_GETDCBINFO

■ ASYNC_SETLINECTRL

USHORT DosDevIOCtl(0L, *pbLineCtrl*, **0x0042, 0x0001,** *hDevice***)**
PBYTE *pbLineCtrl*; /* pointer to structure with line settings */
HFILE *hDevice*; /* device handle */

The ASYNC_SETLINECTRL function sets the line characteristics (stop bits, parity, and data bits) for the specified serial device.

Parameters

pbLineCtrl Points to the **LINECONTROL** structure that contains the settings for the number of data bits, parity, and number of stop bits. The **LINECONTROL** structure has the following form:

```
typedef struct _LINECONTROL {
    BYTE bDataBits;
    BYTE bParity;
    BYTE bStopBits;
    BYTE fbTransBreak;
} LINECONTROL;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if any of the specified line characteristics is out of range. When an error occurs, line characteristics remain unchanged.

Comments

When a device is first opened, the initial line characteristics are 7 data bits, even parity, and 1 stop bit. After line characteristics are changed, they remain changed until the function is used again, even if the device is closed and reopened.

If the number of data bits is less than 8, the device driver fills with zeros the unused high-order bits of each character it receives from the device; the device driver ignores the unused high-order bits of characters it receives from the program. Therefore, if the number of data bits is 7 but the XOFF character is 0x80, the device driver does not recognize the XOFF character even when automatic-transmission control is enabled. If the error substitution character is 0x80, the device driver still places 0x80 in the receive queue. Programs must see that these characters match the specified data size. Any characters that were in the receive queue before the function is called remain unchanged.

See Also

DosOpen, ASYNC_GETLINECTRL

■ ASYNC_SETMODEMCTRL

```
USHORT DosDevIOCtl(pfCommErr, pbCtrlSignals, 0x0046, 0x0001, hDevice)
PUSHORT pfCommErr;      /* pointer to variable for error value      */
PBYTE pbCtrlSignals;    /* pointer to structure with control signals */
HFILE hDevice;          /* device handle                            */
```

The ASYNC_SETMODEMCTRL function sets the modem-control signals. This function turns on or off the data-terminal-ready (DTR) and ready-to-transmit (RTS) signals (initially, the DTR and RTS signals are turned off).

Parameters

pfCommErr Points to the variable that receives the communication status of the device. This variable can be a combination of the following values:

Value	Meaning
RX_QUE_OVERRUN	Receive queue overrun. There is no room in the device driver receive queue to put a character read in from the receive hardware.
RX_HARDWARE_OVERRUN	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.
PARITY_ERROR	The hardware detected a parity error.
FRAMING_ERROR	The hardware detected a framing error.

The function sets the variable to zero if it encounters an error.

pbCtrlSignals Points to the **MODEMSTATUS** structure that contains the settings for the modem-control signals. The **MODEMSTATUS** structure has the following form:

```
typedef struct _MODEMSTATUS {
    BYTE fbModemOn;
    BYTE fbModemOff;
} MODEMSTATUS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if the specified signal settings are invalid. When an error occurs, the signal settings remain unchanged.

Comments

This function must not be used to enable or disable the DTR or RTS signal if the signal is being used for input handshaking or toggling on transmit. Any attempt to do so will cause a "general failure" error.

Although the function copies the communication error status to the variable pointed to by the *pfCommErr* parameter, it does not clear the error.

If the serial device is opened after having been closed, the DTR and RTS signals are set to the values specified by the DTR control mode and the RTS control mode, respectively. For a full description, see the ASYNC_SETDCBINFO function (0x0001,0x0053).

After a serial device has been closed, the device driver turns off the DTR and RTS signals, but only after the device has transmitted all data and has waited for at least as long as it would take to transmit 10 additional characters.

See Also

DosOpen, ASYNC_GETMODEMINPUT, ASYNC_GETMODEMOUTPUT

■ ASYNC_STARTTRANSMIT

USHORT DosDevIOCtl(0L, 0L, 0x0048, 0x0001, ***hDevice*****)**
HFILE *hDevice*; /* device handle */

The ASYNC_STARTTRANSMIT function starts transmission. This function allows data transmission to be resumed by the device driver if data transmission is halted due to the ASYNC_STOPTRANSMIT function (0x0001,0x0047) or due to an XOFF character being received while the device driver is in automatic transmit flow control mode. This function is similar to the device receiving the XON character.

Parameters

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments There may be other reasons why transmission is disabled; transmission may not be resumed. For more information, see the ASYNC_GETCOMMSTATUS function (0x0001,0x0064).

See Also **DosOpen**, ASYNC_GETCOMMSTATUS , ASYNC_STOPTRANSMIT

■ ASYNC_STOPTRANSMIT

```
USHORT DosDevIOCtl( 0L, 0L, 0x0047, 0x0001, hDevice)
HFILE hDevice;      /* device handle */
```

The ASYNC_STOPTRANSMIT function stops the device from transmitting. This function stops data transmission by preventing the device driver from sending additional data to the transmit hardware. This function is similar to the device receiving the XOFF character.

Parameters *hDevice* Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments If automatic-transmission control is enabled, this request causes the device driver to behave exactly as if it received the XOFF character. Transmission can be resumed if an XON character is received by the device driver, if an ASYNC_STARTTRANSMIT (0x0001,0x0048) function is received, or if the device driver is told to disable automatic-transmission control and in the previous state automatic-transmission control was enabled.

If automatic-transmission control is disabled, the ASYNC_STARTTRANSMIT function (0x0001,0x0048) must be called for transmission to resume. If, after this request is received, the device driver is told to enable automatic-transmission control, transmission is still disabled. It can be re-enabled by any of the scenarios discussed above.

There still may be other reasons why transmission may be disabled. For more information, see the ASYNC_GETCOMMSTATUS function (0x0001,0x0064).

See Also **DosOpen**, ASYNC_GETCOMMSTATUS , ASYNC_STARTTRANSMIT

■ ASYNC_TRANSMITIMM

```
USHORT DosDevIOCtl( 0L, pbChar, 0x0044, 0x0001, hDevice)
PBYTE pbChar;      /* pointer to character */
FILE hDevice;      /* device handle        */
```

The ASYNC_TRANSMITIMM function transmits the specified byte immediately.

Parameters *pbChar* Points to the character to be transmitted.

hDevice Identifies the serial device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments

The device driver queues the character as the next character to be transmitted even if there are already characters in the transmit queue.

If automatic-receiving control is enabled, an XON or XOFF character may be transmitted before the requested character.

The function always returns before the character is actually transmitted.

If a character is already waiting to be transmitted immediately, the function returns an error. The ASYNC_GETCOMMSTATUS function (0x0001,0x0064) can be used to determine whether a character is currently waiting to be transmitted immediately.

The device driver will not immediately transmit the character that is waiting to be transmitted immediately if the device driver is not transmitting characters due to modem-control signal-output handshaking or if the device driver is currently transmitting a break.

If the device driver is not transmitting characters due to automatic transmission or receiving control (XON/XOFF) being enabled or due to operating as if an XOFF character had been received, the device driver still transmits a character that is waiting to be transmitted immediately due to this request. An application that requests that the device driver transmit a character immediately if automatic transmission or receiving control is enabled may cause unexpected results to happen to the communications line flow control protocol.

This function is generally used to manually send XON and XOFF characters.

The character waiting to be transmitted immediately is not considered part of the device driver transmit queue and is not flushed due to a flush request. XON/XOFF characters that are automatically transmitted due to automatic-receiving control may or may not be placed ahead of the character waiting to be transmitted immediately. Applications should not be dependent on this ordering.

See Also

DosOpen, ASYNC_GETCOMMSTATUS

■ DEV_FLUSHINPUT

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0001, 0x000B, hDevice)
PBYTE pbCommand;      /* pointer to variable with command */
HFILE hDevice;        /* device handle                    */
```

The DEV_FLUSHINPUT function flushes the input buffer.

Parameters

pbCommand Points to the variable that contains a reserved value. This value must be zero.

hDevice Identifies the device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DEV_FLUSHOUTPUT

DEV_FLUSHOUTPUT

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0002, 0x000B, hDevice)
PBYTE pbCommand;     /* pointer to variable with command */
HFILE hDevice;       /* device handle                    */
```

The DEV_FLUSHOUTPUT function flushes the output buffer.

Parameters

pbCommand Points to the variable that contains a reserved value. This value must be zero.

hDevice Identifies the device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DEV_FLUSHINPUT

DEV_QUERYMONSUPPORT

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0060, 0x000B, hDevice)
PBYTE pbCommand;     /* pointer to variable with command */
HFILE hDevice;       /* device handle                    */
```

The DEV_QUERYMONSUPPORT function queries a device driver for monitor support.

Parameters

pbCommand Points to the variable that contains a reserved value. This value must be zero.

hDevice Identifies the device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the device supports character monitors or an error value if an error occurs.

See Also

DosOpen

DSK_BLOCKREMOVABLE

```
USHORT DosDevIOCtl(pfNonRemovable, pbCommand, 0x0020, 0x0008, hDevice)
PBYTE pfNonRemovable;   /* pointer to removable/nonremovable flag */
PBYTE pbCommand;        /* pointer to variable with command       */
HFILE hDevice;          /* device handle                          */
```

The DSK_BLOCKREMOVABLE function indicates whether the block device is removable.

Parameters

pfNonRemovable Points to the variable that receives the medium type. This variable is 0x0000 if the medium is removable or 0x0001 if it is nonremovable.

pbCommand Points to the variable that contains a reserved value. This value must be zero.

hDevice Identifies the disk-drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**

DSK_FORMATVERIFY

USHORT DosDevIOCtl(**0L,** *pbCommand*, **0x0045, 0x0008,** *hDevice***)**
PBYTE *pbCommand*; /* pointer to structure with command */
HFILE *hDevice*; /* device handle */

The DSK_FORMATVERIFY function formats and verifies a track on a disk drive according to the information passed in the format table. The format table is passed to the controller and the controller performs whatever operations are necessary for formatting.

Parameters *pbCommand* Points to the **TRACKFORMAT** structure that contains information about the format operation. The **TRACKFORMAT** structure has the following form:

```
typedef struct _TRACKFORMAT {
    BYTE bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT reserved;
    USHORT cSectors;
    struct {
        BYTE bCylinder;
        BYTE bHead;
        BYTE idSector;
        BYTE bBytesSector;
    } FormatTable[1];
} TRACKFORMAT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the disk-drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments Some controllers do not support formatting tracks with varying sector sizes. The program must make sure that the sector sizes specified in the format table are all the same.

See Also **DosOpen**

DSK_GETDEVICEPARAMS

USHORT DosDevIOCtl(*pbBPB, pbCommand,* **0x0063, 0x0008,** *hDevice***)**

```
PBYTE pbBPB;        /* pointer to structure for BIOS parameter blocks */
PBYTE pbCommand;    /* pointer to variable with command              */
HFILE hDevice;      /* device handle                                 */
```

The DSK_GETDEVICEPARAMS function retrieves the device parameters for an MS OS/2 block device. The device driver maintains two BIOS parameter blocks (BPB) for each disk drive. One block corresponds to the medium currently in the disk drive. The other is a recommended BPB, based on the type of medium that corresponds to the physical device. For example, a high-density disk drive has a BPB for a 96 tracks-per-inch (tpi) floppy disk; a low-density disk drive has a BPB for a 48-tpi floppy disk.

Parameters

pbBPB Points to the **BIOSPARAMETERBLOCK** structure that receives the BPB. The **BIOSPARAMETERBLOCK** structure has the following form:

```
typedef struct _BIOSPARAMETERBLOCK {
    USHORT usBytesPerSector;
    BYTE   bSectorsPerCluster;
    USHORT usReservedSectors;
    BYTE   cFATs;
    USHORT cRootEntries;
    USHORT cSectors;
    BYTE   bMedia;
    USHORT usSectorsPerFAT;
    USHORT usSectorsPerTrack;
    USHORT cHeads;
    ULONG  cHiddenSectors;
    ULONG  cLargeSectors;
    USHORT cCylinders;
    BYTE   bDeviceType;
    USHORT fDeviceAttr;
} BIOSPARAMETERBLOCK;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to the variable that specifies which BPB to retrieve. If the variable is 0x0000, the function retrieves the recommended BPB for the drive (the BPB for the physical device). If the variable is 0x0001, the function retrieves the BPB for the medium currently in the drive.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DSK_SETDEVICEPARAMS

DSK_GETLOGICALMAP

USHORT DosDevIOCtl(*pbDrive, pbCommand,* **0x0021, 0x0008,** *hDevice***)**

```
PBYTE pbDrive;      /* pointer to variable for drive number */
PBYTE pbCommand;    /* pointer to variable with command     */
HFILE hDevice;      /* device handle                        */
```

The DSK_GETLOGICALMAP function retrieves the mapping of a logical drive.

Parameters

pbDrive Points to the variable that receives the logical-drive number. This can be 1 for drive A, 2 for drive B, and so on. The function sets the variable to zero if only one logical drive is mapped to the physical drive.

pbCommand Points to a variable that contains a reserved value. The value must be zero.

hDevice Identifies the physical device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, SETLOGICALMAP

■ DSK_LOCKDRIVE

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0000, 0x0008, hDevice)
PBYTE pbCommand;    /* pointer to variable with command */
HFILE hDevice;      /* device handle                    */
```

The DSK_LOCKDRIVE function locks a disk drive, preventing file I/O by another process on the volume in the disk drive. This function succeeds if there is only one file handle open on the volume in the disk drive because the desired result is to exclude all other I/O to the volume.

Parameters

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DSK_UNLOCKDRIVE

■ DSK_READTRACK

```
USHORT DosDevIOCtl(pbBuffer, pbCommand, 0x0064, 0x0008, hDevice)
PBYTE pbBuffer;     /* pointer to buffer for data         */
PBYTE pbCommand;    /* pointer to structure with command  */
HFILE hDevice;      /* device handle                      */
```

The DSK_READTRACK function reads from a track on a specified disk drive. The track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are odd-numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is read.

Parameters

pbBuffer Points to the buffer that receives data read from the track.

pbCommand Points to the **TRACKLAYOUT** structure that contains the information about the read operation. The **TRACKLAYOUT** structure has the following form:

```
typedef struct _TRACKLAYOUT {
    BYTE   bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT firstSector;
    USHORT cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The device driver will not correctly read sectors of sizes other than 512 bytes if reading would generate a direct-memory-access (DMA) violation error. Programs must ensure that this error does not occur.

See Also **DosOpen**, DSK_WRITETRACK

■ DSK_REDETERMINEMEDIA

USHORT DosDevIOCtl(**0L,** *pbCommand*, **0x0002, 0x0008,** *hDevice***)**
PBYTE *pbCommand*; /* pointer to variable with command */
HFILE *hDevice*; /* device handle */

The DSK_REDETERMINEMEDIA function redetermines the media on a block device and updates the volume in the drive. This function is normally issued after the volume identification information has been changed (for example, by formatting the disk). This function should be called only if the volume is locked.

Parameters *pbCommand* Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**

DSK_SETDEVICEPARAMS

```
USHORT DosDevIOCtl(pbBPB, pbCommand, 0x0043, 0x0008, hDevice)
PBYTE pbBPB;        /* pointer to structure with BIOS parameter blocks */
PBYTE pbCommand;    /* pointer to buffer with command                  */
HFILE hDevice;      /* device handle                                   */
```

The DSK_SETDEVICEPARAMS function sets the device parameters for an MS OS/2 block device. The device driver maintains two BIOS parameter blocks (BPB) for each disk drive. One block is the BPB that corresponds to the medium currently in the disk drive. The other block is a recommended BPB, based on the type of medium that corresponds to the physical device. For example, a high-density disk drive has a BPB for a 96 tracks per inch (tpi) floppy disk; a low-density disk drive has a BPB for a 48-tpi floppy disk.

Parameters

pbBPB Points to the **BIOSPARAMETERBLOCK** structure that contains the device parameters to be set for the drive. The **BIOSPARAMETERBLOCK** structure has the following form:

```
typedef struct _BIOSPARAMETERBLOCK {
    USHORT usBytesPerSector;
    BYTE   bSectorsPerCluster;
    USHORT usReservedSectors;
    BYTE   cFATs;
    USHORT cRootEntries;
    USHORT cSectors;
    BYTE   bMedia;
    USHORT usSectorsPerFAT;
    USHORT usSectorsPerTrack;
    USHORT cHeads;
    ULONG  cHiddenSectors;
    ULONG  cLargeSectors;
    USHORT cCylinders;
    BYTE   bDeviceType;
    USHORT fDeviceAttr;
} BIOSPARAMETERBLOCK;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Point to the variable that contains the command description. This variable can be one of the following values:

Value	Meaning
BUILD_BPB_FROM_MEDIUM	Build the BIOS parameter block (BPB) from the medium for all subsequent build BPB requests.
REPLACE_BPB_FOR_DEVICE	Change the default BPB for the physical device.
REPLACE_BPB_FOR_MEDIUM	Change the BPB for the medium to the specified BPB. Return the new BPB as the BPB for the medium for all subsequent build BPB requests.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DSK_GETDEVICEPARAMS

DSK_SETLOGICALMAP

```
USHORT DosDevIOCtl(pbDrive, pbCommand, 0x0003, 0x0008, hDevice)
PBYTE pbDrive;        /* pointer to variable with drive number */
PBYTE pbCommand;      /* pointer to variable with command      */
HFILE hDevice;        /* device handle                         */
```

The DSK_SETLOGICALMAP function sets the logical-drive mapping for a block device.

Parameters

pbDrive Points to the variable that contains the logical-drive number. This can be 1 for drive A, 2 for drive B, and so on. When the function returns, it copies the specified drive's current logical-drive number to the variable. If only one logical device is mapped to the physical drive, the function sets the variable to zero.

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DSK_GETLOGICALMAP

DSK_UNLOCKDRIVE

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0001, 0x0008, hDevice)
PBYTE pbCommand;      /* pointer to variable with command */
HFILE hDevice;        /* device handle                    */
```

The DSK_UNLOCKDRIVE function unlocks a drive. The drive requires the locked volume represented by the handle.

Parameters

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, DSK_LOCKDRIVE

DSK_VERIFYTRACK

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0065, 0x0008, hDevice)
PBYTE pbCommand;      /* pointer to structure with command */
HFILE hDevice;        /* device handle                     */
```

The DSK_VERIFYTRACK function verifies an operation on a specified disk drive.

Parameters

pbCommand Points to the **TRACKLAYOUT** structure that contains information about the verification operation. The **TRACKLAYOUT** structure has the following form:

```
typedef struct _TRACKLAYOUT {
    BYTE   bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT firstSector;
    USHORT cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

The track-layout table passed in the function determines the sector number, which is passed to the disk controller. When the sectors are odd-numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is verified.

See Also

DosOpen, DSK_READTRACK, PDSK_VERIFYPHYSTRACK, DSK_WRITETRACK

■ DSK_WRITETRACK

USHORT DosDevIOCtl(*pbBuffer*, *pbCommand*, **0x0044, 0x0008,** *hDevice***)**

```
PBYTE pbBuffer;      /* pointer to buffer with data       */
PBYTE pbCommand;     /* pointer to structure with command */
HFILE hDevice;       /* device handle                     */
```

The DSK_WRITETRACK function writes to a track on a specified disk drive.

Parameters

pbBuffer Points to the buffer that contains the data to be written.

pbCommand Points to the **TRACKLAYOUT** structure that contains information about the write operation. The **TRACKLAYOUT** structure has the following form:

```
typedef struct _TRACKLAYOUT {
    BYTE   bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT firstSector;
    USHORT cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The track-layout table passed in the function determines the sector number, which is passed to the disk controller. When the sectors are odd-numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is written.

See Also **DosOpen**, DSK_READTRACK, PDSK_READPHYSTRACK, PDSK_WRITEPHYSTRACK

■ KBD_CREATE

```
USHORT DosDevIOCtl(0L, pbCommand, 0x005D, 0x0004, hDevice)
PBYTE pbCommand;     /* pointer to buffer with handle and pid */
HFILE hDevice;       /* device handle                         */
```

The KBD_CREATE function allocates memory for a logical keyboard (KCB). This function obtains physical memory for a new logical keyboard. The process ID and a logical-keyboard handle passed by the caller stored in allocated memory for use later by the KBD_SETKCB function. A logical keyboard is not created if the handle is zero.

Parameters *pbCommand* Points to the buffer that contains the value to use as the logical-keyboard handle and the code-page identifier to use with the logical keyboard.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if the logical keyboard cannot be created.

See Also KBD_SETKCB, KBD_DESTROY

■ KBD_DESTROY

```
USHORT DosDevIOCtl(0L, pbCommand, 0x005E, 0x0004, hDevice)
PBYTE pbCommand;     /* pointer to buffer with handle and pid */
HFILE hDevice;       /* device handle                         */
```

The KBD_DESTROY function frees memory for a logical keyboard (KCB). This function searches for the existing logical keyboard that has the specified logical-keyboard handle and process ID combination and frees the physical memory associated with the logical keyboard. No action is taken if the specified handle is zero.

Parameters *pbCommand* Points to the buffer that contains the logical-keyboard handle.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if the logical keyboard identified by the given handle cannot be found.

See Also KBD_CREATE

■ KBD_GETCODEPAGEID

USHORT DosDevIOCtl(*pbCPID*, **0L, 0x0078, 0x0004,** *hDevice***)**
PBYTE *pbCPID*; /* pointer to buffer for code page id */
HFILE *hDevice*; /* device handle */

The KBD_GETCODEPAGEID function retrieves the identifier of the code page being used by the current logical keyboard.

Parameters *pbCPID* Points to the **CPID** structure that receives the code-page identifier. The **CPID** structure has the following form:

```
typedef struct _CPID {
    USHORT idCodePage;
    USHORT Reserved;
} CPID;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comment This function sets the identifier to zero to indicate that PC US 437 is being used.

See Also **KbdGetCp**

■ KBD_GETINPUTMODE

USHORT DosDevIOCtl(*pbInputMode*, **0L, 0x0071, 0x0004,** *hDevice***)**
PBYTE *pbInputMode*; /* pointer to variable for input mode */
HFILE *hDevice*; /* device handle */

The KBD_GETINPUTMODE function retrieves the input mode of the screen group of the active process. The input mode defines whether the following keys are processed as commands or as keystrokes: CONTROL+C, CONTROL+BREAK, CONTROL+S, CONTROL+P, SCROLL LOCK, PRINTSCREEN.

Parameters *pbInputMode* Points to the variable that receives the input mode. If the variable is ASCII_MODE, the keyboard has ASCII input mode. If the variable is BINARY_MODE, the keyboard has binary input mode.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, KBD_SETINPUTMODE

KBD_GETINTERIMFLAG

USHORT DosDevIOCtl(*pfFlags***, 0L, 0x0072, 0x0004,** *hDevice***)**

```
PBYTE pfFlags;     /* pointer to variable for flags */
HFILE hDevice;     /* device handle                 */
```

The KBD_GETINTERIMFLAG function retrieves interim character flags.

Parameters

pfFlags Points to the variable that receives interim flags. If the variable is CONVERSION_REQUEST, the program requested conversion. If it is INTERIM_CHAR, the interim console flag is set.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, KBD_SETINTERIMFLAG

KBD_GETKEYBDTYPE

USHORT DosDevIOCtl(*pbType***, 0L, 0x0077, 0x0004,** *hDevice***)**

```
PBYTE pbType;      /* pointer to structure for keyboard type */
HFILE hDevice;     /* device handle                          */
```

The KBD_GETKEYBDTYPE function retrieves information about the type of keyboard being used.

Parameters

pbType Points to the **KBDTYPE** structure that receives the keyboard type. The **KBDTYPE** structure has the following form:

```
typedef struct _KBDTYPE {
    USHORT usType;
    USHORT reserved1;
    USHORT reserved2;
} KBDTYPE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen

KBD_GETSESMGRHOTKEY

USHORT DosDevIOCtl(*pbHotKeyBuf***,** *pcHotKeys***, 0x0076, 0x0004,** *hDevice***)**

```
PBYTE pbHotKeyBuf;     /* pointer to structure for hot-key information */
PUSHORT pcHotKeys;     /* pointer to variable for hot-key count        */
HFILE hDevice;         /* device handle                                */
```

The KBD_GETSESMGRHOTKEY function retrieves the hot-key information structures for the currently defined hot keys.

Parameters

pbHotKeyBuf Points to the **HOTKEY** structure that receives hot-key information structures. The buffer must be at least as large as the number of structures requested. The **HOTKEY** structure has the following form:

```
typedef struct _HOTKEY {
    USHORT fHotKey;
    UCHAR  scancodeMake;
    UCHAR  scancodeBreak;
    USHORT idHotKey;
} HOTKEY;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pcHotKeys Points to the variable that specifies the number of hot-key information structures to retrieve. If this variable is HOTKEY_MAX_COUNT, the function copies a value to the variable that specifies the maximum number of hot keys the keyboard device driver can support. If this variable is HOTKEY_CURRENT_COUNT, the function copies a value to this variable that specifies the actual number of hot keys currently supported. The function also copies the hot-key information to the buffer pointed to by the *pbHotKeyBuf* parameter.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

If the variable pointed to by *pcHotKeys* is HOTKEY_MAX_COUNT, the function returns the number of currently defined hot keys. The program uses this number to allocate sufficient space to retrieve the actual hot-key information (retrieved by setting the variable to HOTKEY_CURRENT_COUNT).

Programs should retrieve the number of hot keys first, allocate sufficient space for the buffer pointed to by the *pbHotKeyBuf* parameter, then retrieve the hot keys.

See Also

DosOpen, KBD_SETSESMGRHOTKEY

■ KBD_GETSHIFTSTATE

USHORT DosDevIOCtl(*pbShiftState***, 0L, 0x0073, 0x0004,** *hDevice***)**
PBYTE *pbShiftState*; /* pointer to structure for shift state */
HFILE *hDevice*; /* device handle */

The KBD_GETSHIFTSTATE function retrieves the shift state of the default keyboard of the current screen group. The shift state identifies whether the SHIFT, CONTROL, ALT, INSERT, and SYSREQ keys are up or down and whether the SCROLL LOCK, NUMLOCK, CAPSLOCK, and INSERT modes are on.

Parameters

pbShiftState Points to the **SHIFTSTATE** structure that receives the shift state. The **SHIFTSTATE** structure has the following form:

```
typedef struct _SHIFTSTATE {
    USHORT fsState;
    BYTE   fbNLS;
} SHIFTSTATE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The shift state is set by incoming keystrokes. It can also be set by using the KBD_SETSHIFTSTATE function (0x0004, 0x0053).

See Also **DosOpen**, KBD_SETSHIFTSTATE

KBD_PEEKCHAR

USHORT DosDevIOCtl(*pkkiBuffer, pusStatus,* **0x0075, 0x0004,** *hDevice***)**
PKBDKEYINFO *pkkiBuffer*; /* pointer to structure for keystroke */
PUSHORT *pusStatus*; /* pointer to variable for status */
HFILE *hDevice*; /* device handle */

The KBD_PEEKCHAR function retrieves one character data record from the head of the keyboard-input buffer of the screen group of the active process. The character data record is not removed from the keyboard-input buffer.

Parameters *pkkiBuffer* Points to the **KBDKEYINFO** structure that contains keyboard input. The **KBDKEYINFO** structure has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pusStatus Points to the variable that receives the keyboard status. It can be one or both of the following values:

Value	Meaning
KBD_DATA_RECEIVED	Character data record is retrieved. If not set, no character data was retrieved.
KBD_DATA_BINARY	Input mode is binary. If not set, input mode is ASCII.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments If the shift-reporting input mode is enabled, the keystroke information retrieved may specify only a shift-state change and no character input.

See Also **DosOpen**, KBD_READCHAR

KBD_READCHAR

USHORT DosDevIOCtl(*pkkiBuffer, pcRecords,* **0x0074, 0x0004,** *hDevice***)**
PKBDKEYINFO *pkkiBuffer;* /* pointer to structure for keystrokes */
PUSHORT *pcRecords;* /* pointer to variable for record count */
HFILE *hDevice;* /* device handle */

The KBD_READCHAR function retrieves one or more character data records from the keyboard-input buffer for the screen group of the active process.

Parameters

pkkiBuffer Points to the structure that receives the character data records. The structure must be at least as large as the size of an individual record multiplied by the requested number of records to be read. The **KBDKEYINFO** structure has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pcRecords Points to the variable that contains the number of records to be read. When the function returns, it copies the actual number of records retrieved to the variable.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

This function copies the records to the buffer pointed to by the *pkkiBuffer* parameter. The variable pointed to by the *pcRecords* parameter specifies the number of records to copy. The function can copy up to 16 characters.

If the variable pointed to by *pcRecords* is KBD_READ_WAIT, the function waits for the requested number of keystrokes; it blocks the calling process until all records have been read. If the variable is KBD_READ_NOWAIT, the function retrieves any available records (up to the specified number) and returns immediately. When the function returns, it copies the actual number of records retrieved to the variable. It sets the sign bit to 0 if the input mode is ASCII; it sets the sign bit to 1 (0x8000) if the input mode is binary.

See Also

DosOpen, KbdCharIn, KBD_PEEKCHAR

KBD_SETFGNDSCREENGRP

USHORT DosDevIOCtl(0L, *pusScreenGrp,* **0x0055, 0x0004,** *hDevice***)**
PUSHORT *pusScreenGrp;* /* pointer to structure with screen group */
HFILE *hDevice;* /* device handle */

The KBD_SETFGNDSCREENGRP function sets the new foreground screen group. When the keyboard switches to the new screen group, it switches to the shift state, input buffer, and monitor chain defined for that screen group.

This function is reserved for the session manager.

Parameters

pusScreenGrp Points to the **SCREENGROUP** structure that contains the screen-group identifier of the new foreground screen group. The **SCREENGROUP** structure has the following form:

```
typedef struct _SCREENGROUP {
    USHORT idScreenGrp;
    USHORT fTerminate;
} SCREENGROUP;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, KBD_SETSESMGRHOTKEY

■ KBD_SETFOCUS

USHORT DosDevIOCtl(0L, *phkbd*, **0x0057, 0x0004,** *hDevice***)**
PHKBD *phkbd*; /* pointer to logical keyboard handle */
HFILE *hDevice*; /* device handle */

The KBD_SETFOCUS function sets the keyboard focus to the specified logical keyboard.

Parameters

phkbd Points to the logical keyboard handle. The handle must have been created previously by using the **KbdOpen** function.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, KbdOpen

■ KBD_SETINPUTMODE

USHORT DosDevIOCtl(0L, *pbInputMode*, **0x0051, 0x0004,** *hDevice***)**
PBYTE *pbInputMode*; /* pointer to variable with input mode */
HFILE *hDevice*; /* device handle */

The KBD_SETINPUTMODE function sets the input and shift-report modes for the keyboard device driver. The input mode defines whether the following input keys are processed as keystrokes or as commands: CONTROL+C, CONTROL+BREAK, CONTROL+S, CONTROL+P, SCROLL LOCK, PRINTSCREEN.

The shift-report mode defines whether the shift keys are processed as shift keys or as keystrokes.

Parameters

pbInputMode Points to the variable that contains the input mode for the keyboard. If the variable is ASCII_MODE, the input mode is ASCII. If the variable is BINARY_MODE, the input mode is binary. If these values are combined with SHIFT_REPORT_MODE, the function enables the shift-report mode; otherwise, the shift-report mode is disabled.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The default input mode is ASCII. The keyboard device driver maintains an input mode for each screen group.

See Also **DosOpen**, KBD_GETINPUTMODE

■ KBD_SETINTERIMFLAG

```
USHORT DosDevIOCtl(0L, pfFlags, 0x0052, 0x0004, hDevice)
PBYTE pfFlags;    /* pointer to variable with flags */
HFILE hDevice;    /* device handle                  */
```

The KBD_SETINTERIMFLAG function sets the interim character flags.

Parameters

pfFlags Points to the variable that contains the interim flags. If the variable is 0x0020, the program requested conversion. If the variable is 0x0080, the interim character flag is set.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The keyboard device driver maintains the interim character flags for each screen group and passes the interim character flags (with each character data record) to the keyboard monitors. The interim character flags set by this function are not the same as the interim character flags in a character data record.

See Also **DosOpen**, KBD_GETINTERIMFLAG

■ KBD_SETKCB

```
USHORT DosDevIOCtl(0L, phKbd, 0x0058, 0x0004, hDevice)
PHKBD phKbd;      /* logical-keyboard handle */
HFILE hDevice;    /* device handle           */
```

The KBD_SETKCB function binds the specified logical keyboard (KCB) to the physical keyboard for this session.

Parameters

phKbd Points to the handle that identifies the logical keyboard.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **KbdGetFocus**

■ KBD_SETNLS

USHORT DosDevIOCtl(0L, *pbCodePage*, **0x005C, 0x0004,** *hDevice***)**
PBYTE *pbCodePage*; /* pointer to structure with code-page info */
HFILE *hDevice*; /* device handle */

The KBD_SETNLS function installs one of two possible code pages into the device driver and updates entry number one or number two of the code-page control block. Entry zero is the device-driver resident code page.

Parameters

pbCodePage Points to the **CODEPAGEINFO** structure that specifies the translation table and code page to be set. The **CODEPAGEINFO** structure has the following form:

```
typedef struct _CODEPAGEINFO {
    PBYTE pbTransTable;
    USHORT idCodePage;
    USHORT idTable;
} CODEPAGEINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comment

This function is similar to KBD_SETTRANSTABLE (0x0004,0x0050) except it updates different entries in the code-page control block.

See Also

DosOpen, KBD_SETTRANSTABLE, **KbdSetCustCp**

■ KBD_SETSESMGRHOTKEY

USHORT DosDevIOCtl(0L, *pbHotKey*, **0x0056, 0x0004,** *hDevice***)**
PBYTE *pbHotKey*; /* pointer to structure with hot key */
HFILE *hDevice*; /* device handle */

The KBD_SETSESMGRHOTKEY function sets the session-manager hot keys. A new hot key applies to all screen groups. The session manager can define up to 16 hot keys.

Parameters

pbHotKey Points to the **HOTKEY** structure that contains the hot-key information. The **HOTKEY** structure has the following form:

```
typedef struct _HOTKEY {
    USHORT fHotKey;
    UCHAR  scancodeMake;
    UCHAR  scancodeBreak;
    USHORT idHotKey;
} HOTKEY;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments The KBD_SETSESMGRHOTKEY function is successful only if it is performed by the process that initially called the KBD_SETFGNDSCREENGRP function (0x0004, 0x0055).

A hot key can be specified as a combination of shift flags and scan codes, including key combinations such as ALT+ESC. The system detects the hot key when the specified scan code is received. If a hot key has already been defined for a given hot-key identifier, specifying the identifier again replaces the previous definition.

See Also **DosOpen**, KBD_GETSESMGRHOTKEY, KBD_SETFGNDSCREENGRP

■ KBD_SETSHIFTSTATE

USHORT DosDevIOCtl(0L, ***pbShiftState*****, 0x0053, 0x0004,** ***hDevice*****)**

```
PBYTE pbShiftState;   /* pointer to structure with shift state */
HFILE hDevice;        /* device handle                         */
```

The KBD_SETSHIFTSTATE function sets the shift state for the default keyboard in the current screen group. The shift state identifies whether the SHIFT, CONTROL, ALT, INSERT, and SYSREQ keys are up or down and whether the SCROLL LOCK, NUMLOCK, CAPSLOCK, and INSERT modes are on.

Parameters *pbShiftState* Points to the **SHIFTSTATE** structure that contains the shift state. The **SHIFTSTATE** structure has the following form:

```
typedef struct _SHIFTSTATE {
    USHORT fsState;
    BYTE   fbNLS;
} SHIFTSTATE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The system puts the shift state into the character data record built for each incoming keystroke; the shift state then can be used to interpret the meaning of keystrokes. The function sets the shift state to the specified state regardless of the state of the actual keys. The shift remains as set until the user presses or releases the corresponding key.

The keyboard device driver maintains a shift state for each screen group.

See Also **DosOpen**, KBD_GETSHIFTSTATE

■ KBD_SETTRANSTABLE

USHORT DosDevIOCtl(0L, ***pbTransTable*****, 0x0050, 0x0004,** ***hDevice*****)**

```
PBYTE pbTransTable;   /* pointer to translation table */
HFILE hDevice;        /* device handle                */
```

The KBD_SETTRANSTABLE function passes a new translation table to the keyboard translation function. The new table, which overlays the current table, translates subsequent keystrokes.

Parameters

pbTransTable Points to the translation table.

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

The default translation table is U.S. English.

See Also

DosOpen

■ KBD_SETTYPAMATICRATE

```
USHORT DosDevIOCtl(0L, pusRateDelay, 0x0054, 0x0004, hDevice)
PUSHORT pusRateDelay;    /* structure with typamatic rate and delay */
HFILE hDevice;           /* device handle                           */
```

The KBD_SETTYPAMATICRATE function sets the keyboard typamatic rate and delay.

Parameters

pusRateDelay Points to the **RATEDELAY** structure that contains the typamatic rate and delay. The **RATEDELAY** structure has the following form:

```
typedef struct _RATEDELAY {
    USHORT delay;
    USHORT rate;
} RATEDELAY;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, GETTYPAMATICRATE

■ KBD_XLATESCAN

```
USHORT DosDevIOCtl(pkbxl, pidCodePage, 0x0079, 0x0004, hDevice)
PKBDXLATE pkbxl;       /* pointer to structure for scan code */
PBYTE pidCodePage;     /* pointer to code page for translation */
HFILE hDevice;         /* device handle                       */
```

The KBD_XLATESCAN function translates a scan code in a character data record to an ASCII character.

Parameters

pkbxl Points to the **KBDTRANS** structure that contains the scan code to translate. It also receives the character value when the function returns. The **KBDTRANS** structure has the following form:

```
typedef struct _KBDTRANS {
    UCHAR   chChar;
    UCHAR   chScan;
    UCHAR   fbStatus;
    UCHAR   bNlsShift;
    USHORT  fsState;
    ULONG   time;
    USHORT  fsDD;
    USHORT  fsXlate;
    USHORT  fsShift;
    USHORT  sZero;
} KBDTRANS;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pidCodePage Points to a code-page identifier that specifies which code page to use for the translation. The code-page identifier can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

hDevice Identifies the keyboard that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

You may specify a code page to use for translation. Otherwise, the code page of the active keyboard is used. On entry, the **KBDTRANS** structure specifies the code page to use for translation.

See Also

KbdXlate

MON_REGISTERMONITOR

USHORT DosDevIOCtl(*pusInfo*, *pbCommand*, **0x0040, 0x000A,** *hDevice***)**

PUSHORT *pusInfo*; /* pointer to structure with monitor-register info */
PBYTE *pbCommand*; /* pointer to command */
HFILE *hDevice*; /* device handle */

The MON_REGISTERMONITOR function registers a monitor.

Parameters

pusInfo Points to the **MONITORPOSITION** structure that contains the monitor-registration information. The **MONITORPOSITION** structure has the following form:

```
typedef struct _MONITORPOSITION {
    USHORT position;
    USHORT index;
    PBYTE pbInBuf;
    USHORT offset;
} MONITORPOSITION;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosMonRead, DosMonReg, DosMonWrite, DosOpen**

■ MOU_ALLOWPTRDRAW

USHORT DosDevIOCtl(**0L, 0L, 0x0050, 0x0007,** *hDevice***)**
HFILE *hDevice*; /* device handle */

The MOU_ALLOWPTRDRAW function notifies the mouse device driver that the screen group has been switched and that the pointer can now be drawn.

Parameters *hDevice* Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**

■ MOU_DRAWPTR

USHORT DosDevIOCtl(**0L, 0L, 0x0057, 0x0007,** *hDevice***)**
HFILE *hDevice*; /* device handle */

The MOU_DRAWPTR function removes the current exclusion rectangle, allowing the pointer to be drawn anywhere on the screen. If an exclusion rectangle has been declared for the screen group, that rectangle is released and the pointer position is checked. If the pointer was in the released rectangle, it is drawn. If the pointer was not in the released rectangle, the pointer-draw operation occurs.

Parameters *hDevice* Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**

■ MOU_GETBUTTONCOUNT

```
USHORT DosDevIOCtl(pusCount, 0L, 0x0060, 0x0007, hDevice)
PUSHORT pusCount;    /* pointer to variable for button count */
HFILE hDevice;       /* device handle                        */
```

The MOU_GETBUTTONCOUNT function retrieves a count of the number of mouse buttons.

Parameters

pusCount Points to the variable that receives the count mouse buttons.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen

■ MOU_GETEVENTMASK

```
USHORT DosDevIOCtl(pfEvents, 0L, 0x0065, 0x0007, hDevice)
PUSHORT pfEvents;    /* pointer to variable for event mask */
HFILE hDevice;       /* device handle                      */
```

The MOU_GETEVENTMASK function retrieves the event mask of the current pointing device.

Parameters

pfEvents Points to the variable that receives the event mask. This variable can be a combination of the following values:

Value	Meaning
MOUSE_MOTION	Motion; no buttons pressed.
MOUSE_MOTION_WITH_BN1_DOWN	Motion with button 1 pressed.
MOUSE_BN1_DOWN	Button 1 pressed.
MOUSE_MOTION_WITH_BN2_DOWN	Motion with button 2 pressed.
MOUSE_BN2_DOWN	Button 2 pressed.
MOUSE_MOTION_WITH_BN3_DOWN	Motion with button 3 pressed.
MOUSE_BN3_DOWN	Button 3 pressed.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, MOU_SETEVENTMASK

■ MOU_GETHOTKEYBUTTON

```
USHORT DosDevIOCtl(pfHotKey, 0L, 0x0069, 0x0007, hDevice)
PUSHORT pfHotKey;    /* pointer to variable for hot key */
HFILE hDevice;       /* device handle                   */
```

The MOU_GETHOTKEYBUTTON function retrieves the mouse-button equivalent for the system hot key.

Parameters

pfHotKey Points to the variable that receives the hot key. This variable can be one or more of the following values:

Value	Meaning
MHK_NO_HOTKEY	No system hot key used.
MHK_BUTTON1	Button 1 is system hot key.
MHK_BUTTON2	Button 2 is system hot key.
MHK_BUTTON3	Button 3 is system hot key.

If 0x0001 is specified, no system hot-key support is provided. If multiple values are given (excluding 0x0001) the system hot key requires that the indicated buttons be pressed simultaneously.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, MOU_SETHOTKEYBUTTON

■ MOU_GETMICKEYCOUNT

```
USHORT DosDevIOCtl(pcMickeys, 0L, 0x0061, 0x0007, hDevice)
PUSHORT pcMickeys;    /* pointer to variable for mickeys */
HFILE hDevice;        /* device handle                   */
```

The MOU_GETMICKEYCOUNT function retrieves the count of mickeys per centimeter for a given pointing device.

Parameters

pcMickeys Points to the variable that receives the number of mickeys per centimeter. The number can be any value from 0 through 32,767.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen

MOU_GETMOUSTATUS

USHORT DosDevIOCtl(*pfStatus***, 0L, 0x0062, 0x0007,** *hDevice***)**

```
PUSHORT pfStatus;    /* pointer to variable for status flags */
HFILE hDevice;       /* device handle                        */
```

The MOU_GETMOUSTATUS function retrieves the current status flags of the mouse device driver.

Parameters

pfStatus Points to the variable that receives the status flags. This variable can be a combination of the following values:

Value	Meaning
MOUSE_QUEUEBUSY	Event queue is busy with I/O.
MOUSE_BLOCKREAD	Block read is in progress.
MOUSE_FLUSH	Flush is in progress.
MOUSE_UNSUPPORTED_MODE	Pointer-draw routine is disabled (device in unsupported mode).
MOUSE_DISABLED	Interrupt-level pointer-draw routine is not called.
MOUSE_MICKEYS	Mouse data is returned in mickeys (not pels).

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, MOU_SETMOUSTATUS

MOU_GETPTRPOS

USHORT DosDevIOCtl(*pplPosition***, 0L, 0x0067, 0x0007,** *hDevice***)**

```
PPTRLOC pplPosition;   /* pointer to structure for position */
HFILE hDevice;         /* device handle                     */
```

The MOU_GETPTRPOS function retrieves the position of the current screen's pointer.

Parameters

pplPosition Points to the **PTRLOC** structure that receives the new pointer position. The **PTRLOC** structure has the following form:

```
typedef struct _PTRLOC {
        USHORT row;
        USHORT col;
} PTRLOC;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The coordinate values depend on the display mode. If the display is in text mode, character-position values are used. If the display is in graphics mode, pel values are used.

See Also **DosOpen**, MOU_SETPTRPOS

■ MOU_GETPTRSHAPE

```
USHORT DosDevIOCtl(pbBuffer, ppsShape, 0x0068, 0x0007, hDevice)
PBYTE pbBuffer;        /* pointer to buffer for pointer masks       */
PPTRSHAPE ppsShape;    /* pointer to structure for shape information */
HFILE hDevice;         /* device handle                             */
```

The MOU_GETPTRSHAPE function retrieves the current pointer shape.

Parameters *pbBuffer* Points to the buffer that receives the pointer shape. The image format depends on the mode of the display. For currently supported modes, the buffer always consists of the AND image data followed by the XOR image data. The buffer always describes one display plane.

ppsShape Points to the **PTRSHAPE** structure that receives the pointer information and shape. The **PTRSHAPE** structure has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The function exits in a normal state if the input pointer-image buffer is large enough to store the pointer image. The current pointer information is returned in the pointer-data record, and the pointer-image data is copied into the data-packet buffer.

An "invalid buffer size" error occurs if the input pointer-image buffer is smaller than the amount of storage necessary for copying the data. The buffer length returned will be minimum value.

Comments The parameter values are in the same mode as the current screen-group display mode. For text mode, these are character values; for graphics mode, these are pel values.

On input, the only field in the pointer-definition record used by the mouse device driver is the length of the pointer-image buffer.

See Also **DosOpen**, MOU_SETPTRSHAPE

MOU_GETQUESTATUS

USHORT DosDevIOCtl(*pmqiStatus***, 0L, 0x0064, 0x0007,** *hDevice***)**
PMOUQUEINFO *pmqiStatus*; /* pointer to structure for queue status */
HFILE *hDevice*; /* device handle */

The MOU_GETQUESTATUS function retrieves the number of elements in the event queue and the maximum number of elements allowed in an event queue.

Parameters

pmqiStatus Points to the **MOUQUEINFO** structure that receives the queue status. The **MOUQUEINFO** structure has the following form:

```
typedef struct _MOUQUEINFO {
    USHORT cEvents;
    USHORT cmaxEvents;
} MOUQUEINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen

MOU_GETSCALEFACTORS

USHORT DosDevIOCtl(*psfFactors***, 0L, 0x0066, 0x0007,** *hDevice***)**
PSCALEFACT *psfFactors*; /* pointer to structure for scaling factors */
HFILE *hDevice*; /* device handle */

The MOU_GETSCALEFACTORS function retrieves the scaling factors of the current pointing device. Scaling factors are the ratio values that determine how much relative movement is necessary before the mouse device driver reports a pointing-device event. In graphics mode, this ratio is given in mickeys-per-pel. In text mode, this ratio is given in mickeys-per-character. The default values are one mickey-per-row and one mickey-per-column.

Parameters

psfFactors Points to the **SCALEFACT** structure that receives the scaling factors. The **SCALEFACT** structure has the following form:

```
typedef struct _SCALEFACT {
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, MOU_SETSCALEFACTORS

■ MOU_READEVENTQUE

```
USHORT DosDevIOCtl(pmeiEvent, pfWait, 0x0063, 0x0007, hDevice)
PMOUEVENTINFO pmeiEvent;   /* pointer to structure for event information */
PUSHORT pfWait;            /* pointer to wait/no-wait flag               */
HFILE hDevice;             /* device handle                              */
```

The MOU_READEVENTQUE function reads the event queue for the pointing device.

Parameters

pmeiEvent Points to the **MOUEVENTINFO** structure that receives event-queue information. The **MOUEVENTINFO** structure has the following form:

```
typedef struct _MOUEVENTINFO {
    USHORT fs;
    ULONG  Time;
    USHORT row;
    USHORT col;
} MOUEVENTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pfWait Points to the variable that specifies how to read from the queue if no event is available. If the variable is WAIT, the function returns immediately without an event. If the variable is NOWAIT, the function waits until an event is available.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, MouReadEventQue

■ MOU_REMOVEPTR

```
USHORT DosDevIOCtl(0L, pnprBuffer, 0x0058, 0x0007, hDevice)
PNOPTRRECT pnprBuffer;   /* points to structure with exclusion rectangle */
HFILE hDevice;           /* device handle                                */
```

The MOU_REMOVEPTR function specifies the exclusion rectangle to be used by the device driver. The exclusion rectangle specifies an area on the screen where the pointer-draw routine cannot draw the pointer.

Parameters

pnprBuffer Points to the **NOPTRRECT** structure that contains the dimensions of the exclusion rectangle. The **NOPTRRECT** structure has the following form:

```
typedef struct _NOPTRRECT {
    USHORT row;
    USHORT col;
    USHORT cRow;
    USHORT cCol;
} NOPTRRECT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The pointer is not drawn in the exclusion rectangle until a different area is specified by another call of this function.

If the exclusion rectangle is defined as the entire screen, pointer-draw operations are disabled for the entire screen group.

See Also **DosOpen**

■ MOU_SCREENSWITCH

```
USHORT DosDevIOCtl(0L, pbNotify, 0x0052, 0x0007, hDevice)
PBYTE pbNotify;    /* pointer to structure with screen group */
HFILE hDevice;     /* device handle                          */
```

The MOU_SCREENSWITCH function notifies the mouse device driver that the screen group is about to be switched, and then sets a system pointer-draw enable/disable flag. Any pointer drawing is locked until the flag is cleared by using the MOU_ALLOWPTRDRAW function (0x0007, 0x0050).

Parameters *pbNotify* Points to the **SCREENGROUP** structure that contains the notification type and screen-group identifier. The **SCREENGROUP** structure has the following form:

```
typedef struct _SCREENGROUP {
    USHORT idScreenGrp;
    USHORT fTerminate;
} SCREENGROUP;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**

■ MOU_SETEVENTMASK

```
USHORT DosDevIOCtl(0L, pfEvent, 0x0054, 0x0007, hDevice)
PUSHORT pfEvent;   /* pointer to variable for event mask */
HFILE hDevice;     /* device handle                      */
```

The MOU_SETEVENTMASK function sets the event mask of the pointing device.

Parameters

pfEvent Points to the variable that contains the event mask. This variable can be a combination of the following values:

Value	Meaning
MOUSE_MOTION	Motion; no buttons pressed.
MOUSE_MOTION_WITH_BN1_DOWN	Motion with button 1 pressed.
MOUSE_BN1_DOWN	Button 1 pressed.
MOUSE_MOTION_WITH_BN2_DOWN	Motion with button 2 pressed.
MOUSE_BN2_DOWN	Button 2 pressed.
MOUSE_MOTION_WITH_BN3_DOWN	Motion with button 3 pressed.
MOUSE_BN3_DOWN	Button 3 pressed.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, MOU_GETEVENTMASK

■ MOU_SETHOTKEYBUTTON

```
USHORT DosDevIOCtl(0L, pfHotKey, 0x0055, 0x0007, hDevice)
PUSHORT pfHotKey;    /* pointer to variable with hot key */
HFILE hDevice;       /* device handle                    */
```

The MOU_SETHOTKEYBUTTON function sets the mouse-button equivalent for the system hot key.

Parameters

pfHotKey Points to the variable that specifies the hot key. This variable can be a combination of the following values:

Value	Meaning
MHK_NO_HOTKEY	No system hot key used.
MHK_BUTTON1	Button 1 is system hot key.
MHK_BUTTON2	Button 2 is system hot key.
MHK_BUTTON3	Button 3 is system hot key.

If 0x0001 is specified, no system hot-key support is provided. If multiple values are given (excluding 0x0001), the system hot key requires that the indicated buttons be pressed simultaneously.

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments This function can be called only by the process that initially issues it and should be used only by the command shell.

See Also **DosOpen**, MOU_GETHOTKEYBUTTON

MOU_SETMOUSTATUS

USHORT DosDevIOCtl(0L, *pfStatus*, **0x005C, 0x0007,** *hDevice***)**
PUSHORT *pfStatus*; /* pointer to variable with status */
HFILE *hDevice*; /* device handle */

The MOU_SETMOUSTATUS function sets a subset of the current mouse device-driver status flags.

Parameters

pfStatus Points to the variable that contains the status flags for the pointing device. If the variable is MOUSE_DISABLED, the interrupt-level pointer-draw routine is not called. If the variable is MOUSE_MICKEYS, mouse data is returned in mickeys (not pels).

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, MOU_GETMOUSTATUS

MOU_SETPROTDRAWADDRESS

USHORT DosDevIOCtl(0L, *pbFunction*, **0x005A, 0x0007,** *hDevice***)**
PBYTE *pbFunction*; /* pointer to structure with drawing function */
HFILE *hDevice*; /* device handle */

The MOU_SETPROTDRAWADDRESS function notifies the mouse device driver of the address of a protected-mode pointer-draw function. This function is valid for protected mode only.

Parameters

pbFunction Points to the **PTRDRAWFUNCTION** structure that contains the address of the pointer-draw function. The **PTRDRAWFUNCTION** structure has the following form:

```
typedef struct _PTRDRAWFUNCTION {
    PFN    pfnDraw;
    PCH    pchDataSeg;
} PTRDRAWFUNCTION;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments

The pointer-draw routine is an installed, pseudo-character device driver. The mouse handler must do the following:

- Open the pointer-draw device driver.
- Query the pointer-draw device driver for the address of its entry point.
- Pass the resulting address of the pointer-draw entry point to the mouse device driver that uses this function.

See Also

DosOpen, MOU_SETREALDRAWADDRESS

MOU_SETPTRPOS

```
USHORT DosDevIOCtl(0L, pplPosition, 0x0059, 0x0007, hDevice)
PPTRLOC pplPosition;    /* pointer to structure with pointer position */
HFILE hDevice;          /* device handle                              */
```

The MOU_SETPTRPOS function sets a new screen position for the pointer image.

Parameters

pplPosition Points to the **PTRLOC** structure that contains the new position for the pointer. The **PTRLOC** structure has the following form:

```
typedef struct _PTRLOC {
    USHORT row;
    USHORT col;
} PTRLOC;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

The coordinate values depend on the display mode. If the display is in text mode, character-position values are used. If the display is in graphics mode, pel values are used.

This function has no effect on the current exclusion-rectangle definitions. If a pointer image is already defined for the screen group, it is replaced by the new pointer image.

If the pointer image is directed into an existing exclusion rectangle, it remains hidden (invisible) until sufficient movement places the pointer outside the exclusion rectangle or until the exclusion rectangle is released.

See Also

DosOpen, MOU_GETPTRPOS

MOU_SETPTRSHAPE

USHORT DosDevIOCtl(*pbBuffer*, *ppsShape*, **0x0056, 0x0007,** *hDevice***)**
PBYTE *pbBuffer*; /* pointer to structure with shape masks */
PPTRSHAPE *ppsShape*; /* pointer to structure with shape information */
HFILE *hDevice*; /* device handle */

The MOU_SETPTRSHAPE function sets the pointer shape.

Parameters

pbBuffer Points to the buffer that contains the pointer image. The image format depends on the mode of the display. For currently supported modes, the buffer always consists of the AND image data, followed by the XOR image data. The buffer always describes one display plane.

ppsShape Points to the **PTRSHAPE** structure that receives the pointer information and shape. The **PTRSHAPE** structure has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

The parameter values must be in the same mode as the current screen-group display mode. For text mode, these must be character values; for graphics mode, these must be pel values.

See Also

DosOpen, MOU_GETPTRSHAPE

MOU_SETREALDRAWADDRESS

USHORT DosDevIOCtl(0L, *pbFunction*, **0x005B, 0x0007,** *hDevice***)**
PBYTE *pbFunction*; /* pointer to structure with function */
HFILE *hDevice*; /* device handle */

The MOU_SETREALDRAWADDRESS function notifies the real-mode mouse device driver of the entry point of a real-mode pointer-draw routine. This function is intended for use by the session manager at the end of system initialization and is valid for real mode only.

Parameters

pbFunction Points to the **PTRDRAWFUNCTION** structure that contains the address of the pointer-draw function. The **PTRDRAWFUNCTION** structure has the following form:

```
typedef struct _PTRDRAWFUNCTION {
    PFN    pfnDraw;
    PCH    pchDataSeg;
} PTRDRAWFUNCTION;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, MOU_SETPROTDRAWADDRESS

■ MOU_SETSCALEFACTORS

USHORT DosDevIOCtl(0L, *psfFactors*, **0x0053, 0x0007,** *hDevice***)**
PSCALEFACT *psfFactors*; /* pointer to structure with factors */
HFILE *hDevice*; /* device handle */

The MOU_SETSCALEFACTORS function reassigns the scaling factors of the current pointing device. Scaling factors are ratio values that determine how much relative movement is necessary before the mouse device driver reports a pointing-device event. In graphics mode, the ratio is given in mickeys-per-pel. In text mode, the ratio is given in mickeys-per-character. The default ratio values are one mickey-per-row and one mickey-per-column.

Parameters *psfFactors* Points to the **SCALEFACT** structure that contains the scale factors. The **SCALEFACT** structure has the following form:

```
typedef struct _SCALEFACT {
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, MOU_GETSCALEFACTORS

■ MOU_UPDATEDISPLAYMODE

USHORT DosDevIOCtl(0L, *pviomi*, **0x0051, 0x0007,** *hDevice***)**
PVIOMODEINFO *pviomi*; /* pointer to structure with screen mode */
HFILE *hDevice*; /* device handle */

The MOU_UPDATEDISPLAYMODE function notifies the mouse device driver that the display mode has been modified.

Parameters *pviomi* Points to the **VIOMODEINFO** structure that contains the display-mode information. The **VIOMODEINFO** structure has the following form:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR  fmt_ID;
    UCHAR  attrib;
} VIOMODEINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

When the video I/O subsystem or registered video I/O subsystem sets the display mode, it must notify the mouse device driver prior to switching display modes, in order to synchronize the mouse device driver's functions that update the pointer.

See Also

DosOpen, VioSetMode

■ PDSK_GETPHYSDEVICEPARAMS

USHORT DosDevIOCtl(*pbBlock*, *pbCommand*, **0x0063, 0x0009,** *hDevice***)**
PBYTE *pbBlock*; /* pointer to structure for device parameters */
PBYTE *pbCommand*; /* pointer to variable with command */
HFILE *hDevice*; /* device handle */

The PDSK_GETPHYSDEVICEPARAMS function retrieves the device parameters for a physical device. The retrieved parameters apply to the entire physical disk.

Parameters

pbBlock Points to the **DEVICEPARAMETERBLOCK** structure that receives the device parameters. The **DEVICEPARAMETERBLOCK** structure has the following form:

```
typedef struct _DEVICEPARAMETERBLOCK {
    USHORT reserved1;
    USHORT cCylinders;
    USHORT cHeads;
    USHORT cSectorsPerTrack;
    USHORT reserved2;
    USHORT reserved3;
    USHORT reserved4;
    USHORT reserved5;
} DEVICEPARAMETERBLOCK;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the physical device that receives the device-control function. The handle must have been created previously by using the **DosPhysicalDisk** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosPhysicalDisk**

■ PDSK_LOCKPHYSDRIVE

USHORT DosDevIOCtl(0L, *pbCommand*, 0x0000, 0x0009, *hDevice***)**

```
PBYTE pbCommand;    /* pointer to variable with command */
HFILE hDevice;      /* device handle                    */
```

The PDSK_LOCKPHYSDRIVE function locks the physical drive and any of its associated logical units.

Parameters *pbCommand* Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the disk-drive device that receives the device-control function. The handle must have been created previously by using the **DosPhysicalDisk** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosPhysicalDisk**, PDSK_UNLOCKPHYSDRIVE

■ PDSK_READPHYSTRACK

USHORT DosDevIOCtl(*pbBuffer*, *pbCommand*, 0x0064, 0x0009, *hDevice***)**

```
PBYTE pbBuffer;     /* pointer to structure for data     */
PBYTE pbCommand;    /* pointer to structure with command */
HFILE hDevice;      /* device handle                     */
```

The PDSK_READPHYSTRACK function reads from a physical track on the device specified in the request.

Parameters *pbBuffer* Points to the buffer that receives the data to be read.

pbCommand Points to the **TRACKLAYOUT** structure that contains information about the read operation. The **TRACKLAYOUT** structure has the following form:

```
typedef struct _TRACKLAYOUT {
    BYTE   bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT firstSector;
    USHORT cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosPhysicalDisk** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

This function is similar to the DSK_READTRACK function (0x0008, 0x0064) except that I/O is offset from the beginning of the physical drive instead of from the unit number.

The track table passed in the function determines the sector number, which is passed to the disk controller. When the sectors are odd-numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is read.

The device driver will not correctly read sectors of sizes other than 512 bytes if doing so would generate a direct-memory-access (DMA) violation error.

See Also

DosPhysicalDisk, DSK_WRITETRACK, PDSK_VERIFYPHYSTRACK, PDSK_WRITEPHYSTRACK

■ PDSK_UNLOCKPHYSDRIVE

USHORT DosDevIOCtl(0L, *pbCommand*, **0x0001, 0x0009,** *hDevice***)**
PBYTE *pbCommand*;   /* pointer to variable with command */
HFILE *hDevice*;   /* device handle */

The PDSK_UNLOCKPHYSDRIVE function unlocks the physical disk drive and any of its associated logical units and also affects the logical units on the physical disk drive.

Parameters

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosPhysicalDisk** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, **DosPhysicalDisk**, PDSK_LOCKPHYSDRIVE

■ PDSK_VERIFYPHYSTRACK

USHORT DosDevIOCtl(0L, *pbCommand*, 0x0065, 0x0009, *hDevice* **)**
PBYTE *pbCommand*; /* pointer to structure with verification data */
HFILE *hDevice*; /* device handle */

The PDSK_VERIFYPHYSTRACK function verifies I/O on a physical track on the device specified in the request.

Parameters

pbCommand Points to the **TRACKLAYOUT** structure that contains information about the verify operation. The **TRACKLAYOUT** structure has the following form:

```
typedef struct _TRACKLAYOUT {
    BYTE   bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT firstSector;
    USHORT cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the physical device that receives the device-control function. The handle must have been created previously by using the **DosPhysicalDisk** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

This function is similar to the DSK_VERIFYTRACK function (0x0008, 0x0065) except that I/O is offset from the beginning of the physical drive instead of from the unit number.

The track-layout table passed in the function determines the sector number, which is passed to the disk controller. When the sectors are odd-numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is verified.

See Also

DosPhysicalDisk, DSK_VERIFYTRACK, PDSK_READPHYSTRACK, PDSK_WRITEPHYSTRACK

■ PDSK_WRITEPHYSTRACK

USHORT DosDevIOCtl(*pbBuffer*, *pbCommand*, 0x0044, 0x0009, *hDevice* **)**
PBYTE *pbBuffer*; /* pointer to buffer with data */
PBYTE *pbCommand*; /* pointer to structure with command */
HFILE *hDevice*; /* device handle */

The PDSK_WRITEPHYSTRACK function writes to a physical track on the device specified in the request.

Parameters

pbBuffer Points to the buffer that contains the data to be written.

pbCommand Points to the **TRACKLAYOUT** structure that contains information about the write operation. The **TRACKLAYOUT** structure has the following form:

```
typedef struct _TRACKLAYOUT {
    BYTE   bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT firstSector;
    USHORT cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the disk drive that receives the device-control function. The handle must have been created previously by using the **DosPhysicalDisk** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

This function is similar to the DSK_WRITETRACK function (0x0008, 0x0044) except that I/O is offset from the beginning of the physical drive instead of from the unit number.

The track-layout table passed in this function determines the sector number, which is passed to the disk controller. When the sectors are odd-numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is written.

See Also

DosPhysicalDisk, DSK_WRITETRACK, PDSK_READPHYSTRACK, PDSK_VERIFYPHYSTRACK

■ PRT_ACTIVATEFONT

USHORT DosDevIOCtl(*pbFontInfo*, *pbCommand*, **0x0048, 0x0005,** *hDevice***)**
PBYTE *pbFontInfo*; /* pointer to structure for font info */
PBYTE *pbCommand*; /* pointer to byte with command info */
HFILE *hDevice*; /* device handle */

The PRT_ACTIVATEFONT function activates a font for printing.

Parameters

pbFontInfo Points to a **FONTINFO** structure that specifies the font to activate. The **FONTINFO** structure has the following form:

```
typedef struct _FONTINFO {
    USHORT idCodePage;
    USHORT idFont;
} FONTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to a reserved 8-bit value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

PRT_QUERYACTIVEFONT

■ PRT_GETFRAMECTL

USHORT DosDevIOCtl(*pbFrameCtl, pbCommand,* **0x0062, 0x0005,** *hDevice***)**

```
PBYTE pbFrameCtl;    /* pointer to structure for frame settings */
PBYTE pbCommand;     /* pointer to variable with command        */
HFILE hDevice;       /* device handle                           */
```

The PRT_GETFRAMECTL function retrieves frame-control information for a printer.

Parameters

pbFrameCtl Points to the **FRAME** structure that receives the frame-control information. The **FRAME** structure has the following form:

```
typedef struct _FRAME {
    BYTE bCharsPerLine;
    BYTE bLinesPerInch;
} FRAME;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, PRT_SETFRAMECTL

■ PRT_GETINFINITERETRY

USHORT DosDevIOCtl(*pfRetry, pbCommand,* **0x0064, 0x0005,** *hDevice***)**

```
PBYTE pfRetry;       /* pointer to variable for retry flag  */
PBYTE pbCommand;     /* pointer to variable with command    */
HFILE hDevice;       /* device handle                       */
```

The PRT_GETINFINITERETRY function retrieves an infinite retry setting for a printer.

Parameters

pfRetry Points to the variable that receives the infinite retry setting. The variable is FALSE if infinite retry is disabled or TRUE if retry is enabled.

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, PRT_SETINFINITERETRY

■ PRT_GETPRINTERSTATUS

```
USHORT DosDevIOCtl(pfStatus, pbCommand, 0x0066, 0x0005, hDevice)
PBYTE pfStatus;        /* pointer to printer status flag      */
PBYTE pbCommand;       /* pointer to variable with command */
HFILE hDevice;         /* device handle                       */
```

The PRT_GETPRINTERSTATUS function retrieves the status of a printer.

Parameters

pfStatus Points to the variable that receives the printer status. This variable can be a combination of the following values:

Value	Meaning
PRINTER_TIMEOUT	Time-out occurred.
PRINTER_IO_ERROR	I/O error occurred.
PRINTER_SELECTED	Printer selected.
PRINTER_OUT_OF_PAPER	Printer out of paper.
PRINTER_ACKNOWLEDGED	Printer acknowledged.
PRINTER_NOT_BUSY	Printer not busy.

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen

■ PRT_INITPRINTER

```
USHORT DosDevIOCtl(0L, pbCommand, 0x0046, 0x0005, hDevice)
PBYTE pbCommand;       /* command value */
HFILE hDevice;         /* device handle   */
```

The PRT_INITPRINTER function initializes a printer.

Parameters

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen

PRT_QUERYACTIVEFONT

USHORT DosDevIOCtl(*pbFontInfo*, *pbCommand*, **0x0069, 0x0005,** *hDevice***)**

```
PBYTE pbFontInfo;     /* pointer to structure for font information  */
PBYTE pbCommand;      /* pointer to byte with command information   */
HFILE hDevice;        /* device handle                              */
```

The PRT_QUERYACTIVEFONT function determines which code page and font are currently active.

Parameters

pbFontInfo Points to a **FONTINFO** structure that specifies the active font. The **FONTINFO** structure has the following form:

```
typedef struct _FONTINFO {
    USHORT idCodePage;
    USHORT idFont;
} FONTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to a reserved 8-bit value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

PRT_ACTIVATEFONT

PRT_SETFRAMECTL

USHORT DosDevIOCtl(*pbFrameCtl*, *pbCommand*, **0x0042, 0x0005,** *hDevice***)**

```
PBYTE pbFrameCtl;     /* pointer to structure with frame settings */
PBYTE pbCommand;      /* pointer to variable with command         */
HFILE hDevice;        /* device handle                            */
```

The PRT_SETFRAMECTL function sets the frame-control information for a printer.

Parameters

pbFrameCtl Points to the **FRAME** structure that contains the frame-control information. The **FRAME** structure has the following form:

```
typedef struct _FRAME {
    BYTE bCharsPerLine;
    BYTE bLinesPerInch;
} FRAME;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, PRT_GETFRAMECTL

PRT_SETINFINITERETRY

```
USHORT DosDevIOCtl(pfRetry, pbCommand, 0x0044, 0x0005, hDevice)
PBYTE pfRetry;        /* pointer to retry flag              */
PBYTE pbCommand;      /* pointer to variable with command  */
HFILE hDevice;        /* device handle                      */
```

The PRT_SETINFINITERETRY function sets infinite retry for a printer.

Parameters

pfRetry Points to the variable that specifies whether to enable infinite retry. If the variable is FALSE, the function disables infinite retry. If the variable is TRUE, the function enables infinite retry.

pbCommand Points to the variable that contains a reserved value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, PRT_GETINFINITERETRY

PRT_VERIFYFONT

```
USHORT DosDevIOCtl(pbFontInfo, pbCommand, 0x006A, 0x0005, hDevice)
PBYTE pbFontInfo;     /* points to structure for font info  */
PBYTE pbCommand;      /* points to byte with command info   */
HFILE hDevice;        /* device handle                      */
```

The PRT_VERIFYFONT function verifies that a particular code page and font are available for the specified printer.

Parameters

pbFontInfo Points to the **FONTINFO** structure that receives information for the available font. The **FONTINFO** structure has the following form:

```
typedef struct _FONTINFO {
    USHORT idCodePage;
    USHORT idFont;
} FONTINFO;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

pbCommand Points to a reserved 8-bit value. The value must be zero.

hDevice Identifies the printer that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

See Also

DosOpen, PRT_ACTIVATEFONT

■ PTR_GETPTRDRAWADDRESS

USHORT DosDevIOCtl(*pbFunctionInfo*, **0L, 0x0072, 0x0003,** *hDevice***)**
PBYTE *pbFunctionInfo*; /* pointer to structure for function */
HFILE *hDevice*; /* device handle */

The PTR_GETPTRDRAWADDRESS function retrieves the entry-point address and other information for the pointer-draw function (the function that draws the mouse pointer on the screen).

Parameters

pbFunctionInfo Points to the **PTRDRAWFUNCTION** structure that receives the function information. The **PTRDRAWFUNCTION** structure has the following form:

```
typedef struct _PTRDRAWFUNCTION {
    PFN    pfnDraw;
    PCH    pchDataSeg;
} PTRDRAWFUNCTION;
```

For a full description, see Chapter 4, "Types, Macros, Structures."

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or an error value if an error occurs.

Comments

The mouse device driver uses the pointer-draw function to update the pointer image on the screen, and retrieves the address and saves it to use whenever the pointer moves.

See Also

DosOpen

Chapter

4

Types, Macros, Structures

4.1 Introduction

This chapter describes the types, macros, and structures used with MS OS/2 **Dos**, **Kbd**, **Mou**, and **Vio** functions. The MS OS/2 functions use many types, macros, and structures that are not part of the standard C language. These types, macros, and structures have been defined to make the task of creating MS OS/2 programs easier and to make program sources clearer and easier to understand.

All types, macros, and structures in this manual are defined in the MS OS/2 C-language include files. Programmers may also wish to use these when developing MS OS/2 programs in other computer languages, such as Pascal or assembly-language. If include files for a given language are not available, a programmer can translate the definitions given in this chapter by following these guidelines:

- Numbers must be integers or fixed-point real numbers. MS OS/2 functions do not support floating-point numbers. An MS OS/2 program can use floating-point numbers if an appropriate run-time library or coprocessor is supplied and if floating-point numbers are not used as parameters to the MS OS/2 functions.
- Structures must be packed. Some compilers align each new field in a structure on word or double-word boundaries. This may leave unused bytes in a structure if a given field is smaller than the width between boundaries. MS OS/2 functions require that unused bytes be removed from structures.
- Reserved fields in structures should be set to zero. Unless otherwise specified, MS OS/2 functions require that reserved fields be set to zero to avoid compatibility problems with future releases of MS OS/2.
- Variable-length structures must be supported. Several MS OS/2 functions use variable-length structures to receive and/or return information. In a variable-length structure, the number of fields varies depending on when the structure is used. In the C language, programs typically support variable-length structures by allocating enough memory for the current number of fields and accessing those fields by using a pointer to the structure. Programs in other languages may use this method or devise their own method for supporting variable-length structures.
- All 16-bit pointers must be relative to an explicitly defined segment register. Some compilers assume that the **ds** and **ss** registers contain the same value and implicitly use one segment for both. MS OS/2 does not guarantee that the **ds** and **ss** registers will be equal. This is especially true in dynamic-link libraries and programs that use callback functions (for example, window procedures).
- All 32-bit pointers must consist of a *selector:offset* pair. A physical address, that is, an address that represents a 32-bit offset from the beginning of physical memory, cannot be used by MS OS/2 functions. (One exception to this rule is the **VioGetPhysBuf** function, which requires a physical address to video memory.)

4.2 Types

The following is a complete list, in alphabetical order, of the types that have been defined for the functions described in this manual. Many of these types begin with a letter that identifies what the type is used for—for example, **H** identifies a handle; **P**, a far pointer; **NP**, a near pointer; and **U**, an unsigned variable.

Type	Meaning
BOOL	16-bit Boolean value.
BYTE	8-bit unsigned value.
CHAR	8-bit signed value.
COLOR	32-bit signed value used to hold a color value.
ERRORID	32-bit value used as an error identifier.
FALSE	Predefined constant set to zero.
HDC	32-bit value used as a device-context handle.
HDIR	16-bit value used as a directory handle.
HFILE	16-bit value used as a file handle.
HKBD	16-bit value used as a logical-keyboard handle.
HMF	32-bit value used as a metafile handle.
HMODULE	16-bit value used as a module handle.
HMONITOR	16-bit value used as a monitor handle.
HMOU	16-bit value used as a mouse handle.
HPIPE	16-bit value used as a pipe handle.
HPS	32-bit value used as a presentation-space handle.
HQUEUE	16-bit value used as a queue handle.
HRGN	16-bit value used as a region handle.
HSEM	32-bit value used as a semaphore handle.
HSYSSEM	32-bit value used as a system semaphore handle.
HTIMER	16-bit value used as a timer handle.
HVIO	16-bit value used as a video-device handle.
INT	16-bit signed value.
LONG	32-bit signed value.
NPBYTE	16-bit pointer to an 8-bit unsigned value.
NPCH	16-bit pointer to a value or array of values.
NPFN	16-bit pointer to a function with **pascal** calling type.

Type	Meaning
NPSZ	16-bit pointer to a null-terminated string.
NULL	Predefined null-pointer value set to zero.
PBOOL	32-bit pointer to a Boolean value.
PBYTE	32-bit pointer to an 8-bit unsigned value.
PCH	32-bit pointer to a value or array of values.
PCHAR	32-bit pointer to a value or array of values.
PCOLOR	32-bit pointer to a color value.
PERRORID	32-bit pointer to an error identifier.
PFN	32-bit pointer to a function with **pascal** calling type.
PFNSIGHANDLER	32-bit pointer to a function with **pascal** calling type.
PHDC	32-bit pointer to a device-context handle.
PHDIR	32-bit pointer to a directory handle.
PHFILE	32-bit pointer to a file handle.
PHKBD	32-bit pointer to a logical-keyboard handle.
PHMF	32-bit pointer to a metafile handle.
PHMODULE	32-bit pointer to a module handle.
PHMONITOR	32-bit pointer to a monitor handle.
PHMOU	32-bit pointer to a mouse handle.
PHPIPE	32-bit pointer to a pipe handle.
PHPS	32-bit pointer to a presentation-space handle.
PHQUEUE	32-bit pointer to a queue handle.
PHRGN	32-bit pointer to a region handle.
PHSEM	32-bit pointer to a semaphore handle.
PHSYSSEM	32-bit pointer to a system-semaphore handle.
PHTIMER	32-bit pointer to a timer handle.
PHVIO	32-bit pointer to a video-device handle.
PID	16-bit value used to hold a process identifier.
PINT	32-bit pointer to a 16-bit signed value.
PLONG	32-bit pointer to a 32-bit signed value.
PPID	32-bit pointer to a process identifier.
PSEL	32-bit pointer to a selector.

Type	Meaning
PSHORT	32-bit pointer to a 16-bit signed value.
PSZ	32-bit pointer to a null-terminated string.
PTID	32-bit pointer to a thread identifier.
PUCHAR	32-bit pointer to an unsigned value or array of values.
PUINT	32-bit pointer to a 16-bit unsigned value.
PULONG	32-bit pointer to a 32-bit unsigned value.
PUSHORT	32-bit pointer to a 16-bit unsigned value.
PVOID	32-bit pointer to an unspecified data type.
SEL	16-bit value used to hold a segment selector.
SHORT	16-bit signed value.
TID	16-bit value used to hold a thread identifier.
TRUE	Predefined constant set to 1.
UCHAR	8-bit unsigned value.
UINT	16-bit unsigned value.
ULONG	32-bit unsigned value.
USHORT	16-bit unsigned value.

4.3 Macros

The following is a complete list, in alphabetical order, of the macros that can be used with the functions described in this manual.

DEFINEMUXSEMLIST

DEFINEMUXSEMLIST(*name, size*)

The **DEFINEMUXSEMLIST** macro creates a structure that is used to hold the semaphore list for the **DosMuxSemWait** function.

Parameters

name Specifies the name of the structure to be created.

size Specifies the size of the structure; that is, the number of semaphores in the list.

See Also

DosMuxSemWait

FIELDOFFSET

FIELDOFFSET(*type, field*)

The **FIELDOFFSET** macro computes the address offset of the specified field in the structure specified by the *type* parameter.

Parameters

type Specifies the name of the structure.

field Specifies the name of a field defined within the given structure.

HIBYTE

HIBYTE(*w*)

The **HIBYTE** macro retrieves the high-order unsigned byte from the 16-bit value specified by the *w* parameter.

Parameters

w Specifies a 16-bit value.

See Also

HIUCHAR, LOBYTE

HIUCHAR

HIUCHAR(*w*)

The **HIUCHAR** macro retrieves the high-order unsigned byte from the 16-bit value specified by the *w* parameter.

Parameters

w Specifies a 16-bit value.

See Also

HIBYTE, LOUCHAR

HIUSHORT

HIUSHORT(*l*)

The **HIUSHORT** macro retrieves the high-order, unsigned 16-bit word from the 32-bit value specified by the *l* parameter.

Parameters *l* Specifies a 32-bit value.

See Also **LOUSHORT**

LOBYTE

LOBYTE(*w*)

The **LOBYTE** macro retrieves the low-order byte from the 16-bit value specified by the *w* parameter.

Parameters *w* Specifies a 16-bit value.

See Also **HIBYTE, LOUCHAR**

LOUCHAR

LOUCHAR(*w*)

The **LOUCHAR** macro retrieves the low-order unsigned byte from the 16-bit value specified by the *w* parameter.

Parameters *w* Specifies a 16-bit value.

See Also **HIUCHAR, LOBYTE**

LOUSHORT

LOUSHORT(*l*)

The **LOUSHORT** macro retrieves the low-order unsigned 16-bit word from the 32-bit value specified by the *l* parameter.

Parameters *l* Specifies a 32-bit value.

See Also **HIUSHORT**

MAKELONG

MAKELONG(*l*, *h*)

The **MAKELONG** macro combines two 16-bit word values to create a 32-bit long integer.

Parameters

l Specifies the low-order 16-bit word value for the new integer.

h Specifies the high-order 16-bit word value for the new integer.

See Also **MAKESHORT, MAKEULONG**

MAKEP

MAKEP(*sel*, *off*)

The **MAKEP** macro combines a segment selector and an address offset to create a far (32-bit) pointer to a memory address.

Parameters

sel Specifies a segment selector. It must be a valid segment selector—for example, if it were created by using the **DosAllocSeg** function.

off Specifies an offset from the beginning of the given segment to the desired byte. The offset must specify an address within the segment.

See Also

DosAllocSeg, OFFSETOF, SELECTOROF

MAKEPGINFOSEG

MAKEPGINFOSEG(*sel*)

The **MAKEPGINFOSEG** macro creates a far (32-bit) pointer to the first byte in the global information segment. The macro assumes that the selector specified by the *sel* parameter has been retrieved by using the **DosGetInfoSeg** function.

Parameters

sel Specifies the segment selector of the global information segment.

Example

```
SEL selGlobalSeg, selLocalSeg;
GINFOSEG FAR *pgis;
DosGetInfoSeg(&selGlobalSeg, &selLocalSeg);
pgis = MAKEPGINFOSEG(selGlobalSeg);
```

See Also

DosGetInfoSeg, MAKEPLINFOSEG

MAKEPLINFOSEG

MAKEPLINFOSEG(*sel*)

The **MAKEPLINFOSEG** macro creates a far (32-bit) pointer to the first byte in the local information segment. The macro assumes that the selector specified by the *sel* parameter has been retrieved by using the **DosGetInfoSeg** function.

Parameters

sel Specifies the segment selector of the local information segment.

Example

```
SEL selGlobalSeg, selLocalSeg;
LINFOSEG FAR *plis;
DosGetInfoSeg(&selGlobalSeg, &selLocalSeg);
lgis = MAKEPGINFOSEG(selGlobalSeg);
```

See Also

DosGetInfoSeg, MAKEPGINFOSEG

MAKESHORT

MAKESHORT(*l*, *h*)

The **MAKESHORT** macro combines two 8-bit values to create a 16-bit integer.

Parameters

l Specifies the low-order 8-bit value of the new integer.

h Specifies the high-order 8-bit value of the new integer.

See Also

MAKELONG, MAKEUSHORT

MAKETYPE

MAKETYPE(*v*, *type*)

The **MAKETYPE** macro casts the variable specified by the *v* parameter as a variable having the type specified by the *type* parameter. This macro permits the contents of the variable to be accessed as if the variable had the specified type.

Parameters

v Specifies the name of the variable to be cast.

type Specifies the name of the data type for the cast.

MAKEULONG

MAKEULONG(*l*, *h*)

The **MAKEULONG** macro combines two 16-bit values to create a 32-bit unsigned integer.

Parameters

l Specifies the low-order 16-bit value of the new integer.

h Specifies the high-order 16-bit value of the new integer.

See Also

MAKELONG, MAKEUSHORT

MAKEUSHORT

MAKEUSHORT(*l*, *h*)

The **MAKEUSHORT** macro combines two 8-bit values to create a 16-bit unsigned integer.

Parameters

l Specifies the low-order 8-bit value of the new integer.

h Specifies the high-order 8-bit value of the new integer.

See Also

MAKESHORT, MAKEULONG

OFFSETOF

OFFSETOF(*p*)

The **OFFSETOF** macro retrieves the address offset of the specified far pointer.

Parameters *p* Specifies a far (32-bit) pointer.

See Also **SELECTOROF**

SELECTOROF

SELECTOROF(*p*)

The **SELECTOROF** macro retrieves the selector from the specified far pointer.

Parameters *p* Specifies a far (32-bit) pointer.

See Also **OFFSETOF**

4.4 Structures

The following is a complete list, in alphabetical order, of the structures used by the functions described in this manual.

■ BIOSPARAMETERBLOCK

```
typedef struct _BIOSPARAMETERBLOCK {     /* bspblk */
    USHORT usBytesPerSector;
    BYTE   bSectorsPerCluster;
    USHORT usReservedSectors;
    BYTE   cFATs;
    USHORT cRootEntries;
    USHORT cSectors;
    BYTE   bMedia;
    USHORT usSectorsPerFAT;
    USHORT usSectorsPerTrack;
    USHORT cHeads;
    ULONG  cHiddenSectors;
    ULONG  cLargeSectors;
    BYTE   abReserved[6];
    USHORT cCylinders;
    BYTE   bDeviceType;
    USHORT fsDeviceAttr;
} BIOSPARAMETERBLOCK;
```

The **BIOSPARAMETERBLOCK** structure contains BIOS parameter blocks.

Fields

usBytesPerSector Specifies the bytes per sector.

bSectorsPerCluster Specifies the sectors per cluster.

usReservedSectors Specifies the reserved sectors.

cFATs Specifies the number of file-allocation tables.

cRootEntries Specifies the maximum number of entries in the root directory.

cSectors Specifies the number of sectors.

bMedia Specifies the media descriptor.

usSectorsPerFAT Specifies the number of sectors per file-allocation table.

usSectorsPerTrack Specifies the number of sectors per track.

cHeads Specifies the number of heads.

cHiddenSectors Specifies the number of hidden sectors.

cLargeSectors Specifies the number of large sectors.

abReserved[6] Specifies six reserved bytes. These must be zero.

cCylinders Specifies the number of cylinders defined for the device.

bDeviceType Specifies the type of device. It can be one of the following values:

Value	Meaning
DEVTYPE_48TPI	48 tracks-per-inch, low-density floppy-disk drive
DEVTYPE_96TPI	96 tracks-per-inch, high-density floppy-disk drive
DEVTYPE_35	3.5-inch (720K) floppy-disk drive
DEVTYPE_8SD	8-inch, single-density floppy-disk drive
DEVTYPE_8DD	8-inch, double-density floppy-disk drive
DEVTYPE_FIXED	Fixed disk
DEVTYPE_TAPE	Tape drive
DEVTYPE_UNKNOWN	Other (unknown type of device)

fsDeviceAttr Specifies information about the drive. If this value is 0x0001, the media are *not* removable. If it is 0x0002, the media can detect changes. This field can be one or both of these values.

See Also DSK_GETDEVICEPARAMS, DSK_SETDEVICEPARAMS

CODEPAGEINFO

```
typedef struct _CODEPAGEINFO {    /* cpi */
    PBYTE pbTransTable;
    USHORT idCodePage;
    USHORT idTable;
} CODEPAGEINFO;
```

The **CODEPAGEINFO** structure specifies the code page and the translation table to be set.

Fields

pbTransTable Points to the keyboard translation table.

idCodePage Specifies a code-page identifier. It can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

idTable Specifies the translation table to be replaced. If this value is 0xFFFF, it specifies the custom translation table.

See Also KBD_SETNLS

COUNTRYCODE

```
typedef struct _COUNTRYCODE {    /* ctryc */
    USHORT country;
    USHORT codepage;
} COUNTRYCODE;
```

The **COUNTRYCODE** structure contains the country code and code-page identifier.

Fields

country Specifies the country code. It can be one of the following values:

Country code	Country
001	United States
002	Canada (French)
003	Latin America
031	Netherlands
032	Belgium

Country code	Country
033	France
034	Spain
039	Italy
041	Switzerland (French)
041	Switzerland (German)
044	United Kingdom
045	Denmark
046	Sweden
047	Norway
049	Germany
061	Australia
351	Portugal
358	Finland

If this field is zero, the function uses the current country code.

codepage Specifies the code-page identifier. It can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

If this field is zero, the function uses the current code-page identifier.

See Also **DosCaseMap, DosGetCollate, DosGetCtryInfo, DosGetDBCSEv**

■ COUNTRYINFO

```
typedef struct _COUNTRYINFO {    /* ctryi */
    USHORT country;
    USHORT codepage;
    USHORT fsDateFmt;
    CHAR   szCurrency[5];
    CHAR   szThousandsSeparator[2];
    CHAR   szDecimal[2];
    CHAR   szDateSeparator[2];
    CHAR   szTimeSeparator[2];
    UCHAR  fsCurrencyFmt;
    UCHAR  cDecimalPlace;
    UCHAR  fsTimeFmt;
    USHORT abReserved1[2];
    CHAR   szDataSeparator[2];
    USHORT abReserved2[5];
} COUNTRYINFO;
```

The **COUNTRYINFO** structure contains country-dependent formatting information.

Fields

country Specifies the country code. It can be one of the following values:

Country code	Country
001	United States
002	Canada (French)
003	Latin America
031	Netherlands
032	Belgium
033	France
034	Spain
039	Italy
041	Switzerland (French)
041	Switzerland (German)
044	United Kingdom
045	Denmark
046	Sweden
047	Norway
049	Germany
061	Australia
351	Portugal
358	Finland

codepage Specifies a reserved value; must be zero.

fsDateFmt Specifies the date format. It can be one of the following values:

Value	Meaning
DATEFMT_MM_DD_YY	Month, day, year (mm/dd/yy)
DATEFMT_DD_MM_YY	Day, month, year (dd/mm/yy)
DATEFMT_YY_MM_DD	Year, month, day (yy/mm/dd)

szCurrency[5] Specifies the currency indicator. It is a null-terminated string.

szThousandsSeparator[2] Specifies the thousands separator. It is a null-terminated string.

szDecimal[2] Specifies the decimal separator. It is a null-terminated string.

szDateSeparator[2] Specifies the date separator. It is a null-terminated string.

szTimeSeparator[2] Specifies the time separator. It is a null-terminated string.

fsCurrencyFmt Specifies the currency format. It can be any combination of the following values:

Value	Meaning
CURRENCY_FOLLOW	Currency indicator follows the money value. If this value is not given, the currency indicator precedes the money value.

Value	Meaning
CURRENCY_SPACE	One space appears between the currency indicator and the money value. If this value is not given, no space appears between the currency indicator and the money value.
CURRENCY_DECIMAL	Specified currency indicator replaces the decimal indicator. If this value is given, other **fsCurrencyFmt** values are ignored.

cDecimalPlace Specifies the number of decimal places (in binary) used in the currency value.

fsTimeFmt Specifies the time format for file directory presentation. If this field is 0x0001, the time is presented in 24-hour (military-time) format. Otherwise, time is presented in a 12-hour format, with "a" and "p" used for A.M. and P.M. indicators.

abReserved1[2] Specifies a reserved value; must be zero.

szDataSeparator[2] Specifies a data-list separator. It is a null-terminated string.

abReserved2[5] Specifies a reserved value; must be zero.

See Also **DosGetCtryInfo**

CPID

```
typedef struct _CPID {      /* cpid */
    USHORT idCodePage;
    USHORT Reserved;
} CPID;
```

The **CPID** structure specifies the code-page identifier for a logical keyboard.

Fields

idCodePage Specifies the code-page ID. It can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

Reserved Specifies a reserved value; must be zero.

See Also KBD_GETCODEPAGEID

DATETIME

```
typedef struct _DATETIME {    /* date */
    UCHAR   hours;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    SHORT   timezone;
    UCHAR   weekday;
} DATETIME;
```

The **DATETIME** structure contains the date and time.

Fields

hours Specifies the current hour using values from 0 through 23.

minutes Specifies the current minute using values from 0 through 59.

seconds Specifies the current second using values from 0 through 59.

hundredths Specifies the current hundredths of a second using values from 0 through 99.

day Specifies the current day of the month using values from 1 through 31.

month Specifies the current month of the year using values from 1 through 12.

year Specifies the current year.

timezone Specifies the difference (in minutes) between the current time zone and Greenwich Mean Time (GMT). This field is positive for time zones west of Greenwich; it is negative for time zones east of Greenwich. For example, for Eastern Standard Time this field is 300 (that is, five hours, 5×60, after GMT). If this field is -1, the time zone is undefined.

weekday Specifies the current day of the week using values from 0 through 6 (Sunday equals zero).

See Also

DosGetDateTime, DosSetDateTime

DCBINFO

```
typedef struct _DCBINFO {    /* dcbinf */
    USHORT usWriteTimeout;
    USHORT usReadTimeout;
    BYTE   fbCtlHndShake;
    BYTE   fbFlowReplace;
    BYTE   fbTimeout;
    BYTE   bErrorReplacementChar;
    BYTE   bBreakReplacementChar;
    BYTE   bXONChar;
    BYTE   bXOFFChar;
} DCBINFO;
```

The **DCBINFO** structure holds device-control block information.

Fields

usWriteTimeout Specifies the time-out in one-hundredths of a second. If set to zero, the time-out is 0.01 seconds; if set to 1, the time-out is 0.02 seconds, and so on.

usReadTimeout Specifies the time-out in one-hundredths of a second. If set to zero, the time-out is 0.01 seconds; if set to 1, the time-out is 0.02 seconds, and so on.

fbCtlHndShake Specifies the control and handshaking modes for the DTR and other signals. It can be a combination of the following values:

Value	Meaning
MODE_DTR_CONTROL	Enable the data-terminal-ready (DTR) control mode.
MODE_DTR_HANDSHAKE	Enable the data-terminal-ready (DTR) input handshaking mode.
MODE_CTS_HANDSHAKE	Enable output handshaking using the clear-to-send (CTS) signal.
MODE_DSR_HANDSHAKE	Enable output handshaking using the data-set-ready (DSR) signal.
MODE_DCD_HANDSHAKE	Enable output handshaking using the data-carrier-detect (DCD) signal.
MODE_DSR_SENSITIVITY	Enable input sensitivity using the data-set-ready (DSR) signal.

fbFlowReplace Specifies the flow control and replacement character modes. It can be a combination of the following values:

Value	Meaning
MODE_AUTO_TRANSMIT	Enable automatic transmit flow control (XON/XOFF).
MODE_AUTO_RECEIVE	Enable automatic receive flow control (XON/XOFF).
MODE_ERROR_CHAR	Enable error replacement character.
MODE_NULL_STRIPPING	Enable null stripping (remove null bytes).
MODE_BREAK_CHAR	Enable break replacement character.
MODE_RTS_CONTROL	Enable the request-to-send (RTS) control mode.
MODE_RTS_HANDSHAKE	Enable the request-to-send (RTS) input handshaking mode.
MODE_TRANSMIT_TOGGLE	Enable toggling on transmit mode.

fbTimeout Specifies the time-out processing for the device. It can be a combination of the following values:

Value	Meaning
MODE_NO_WRITE_TIMEOUT	Enable write infinite time-out processing.
MODE_READ_TIMEOUT	Enable normal read time-out processing.
MODE_WAIT_READ_TIMEOUT	Enable wait-for-something read time-out processing.
MODE_NOWAIT_READ_TIMEOUT	Enable no-wait read time-out processing.

bErrorReplacementChar Specifies the error replacement character.

bBreakReplacementChar Specifies the break replacement character.

bXONChar Specifies the transmission on (XON) character.

bXOFFChar Specifies the transmission off (XOFF) character.

See Also ASYNC_GETDCBINFO, ASYNC_SETDCBINFO

DEVICEPARAMETERBLOCK

```
typedef struct _DEVICEPARAMETERBLOCK {    /* dvpblck */
    USHORT reserved1;
    USHORT cCylinders;
    USHORT cHeads;
    USHORT cSectorsPerTrack;
    USHORT reserved2;
    USHORT reserved3;
    USHORT reserved4;
    USHORT reserved5;
} DEVICEPARAMETERBLOCK;
```

The **DEVICEPARAMETERBLOCK** structure contains device parameters for the physical disk.

Fields

reserved1 Specifies a reserved value; must be zero.

cCylinders Specifies the number of cylinders on the physical device.

cHeads Specifies the number of heads on the physical device.

cSectorsPerTrack Specifies the number of sectors per track on the physical device.

reserved2-reserved5 Specifies a reserved value; must be zero.

See Also PDSK_GETPHYSDEVICEPARAMS

DOSFSRSEM

```
typedef struct _DOSFSRSEM {    /* dosfsrs */
    USHORT cb;
    PID    pid;
    TID    tid;
    USHORT cUsage;
    USHORT client;
    ULONG  sem;
} DOSFSRSEM;
```

The **DOSFSRSEM** structure contains information for a fast-safe RAM semaphore.

Fields

cb Specifies the length of the structure (in bytes). It must be set to 14.

pid Specifies the process identifier of the process that owns the semaphore. If this field is zero, the semaphore is not owned.

tid Specifies the thread identifier of the thread that owns the semaphore.

cUsage Specifies the number of times the owner has issued a **DosFSRamSemRequest** function without a corresponding **DosFSRamSemClear** function.

client Specifies any owner-recorded information that may be needed through maintain the semaphore and the resource being managed.

sem Specifies the RAM semaphore to be used in this request.

See Also **DosFSRamSemClear, DosFSRamSemRequest**

FDATE

```
typedef struct _FDATE {     /* fdate */
    unsigned day      : 5;
    unsigned month    : 4;
    unsigned year     : 7;
} FDATE;
```

The **FDATE** structure is used in various other structures to specify the day, month, and year.

Fields

day Specifies the day.

month Specifies the month.

year Specifies the year.

See Also **FILEFINDBUF, FILESTATUS, FSINFO**

FILEFINDBUF

```
typedef struct _FILEFINDBUF {     /* findbuf */
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
    UCHAR  cchName;
    CHAR   achName[13];
} FILEFINDBUF;
```

The **FILEFINDBUF** structure contains information about a file.

Fields

fdateCreation Specifies the date the file was created.

ftimeCreation Specifies the time the file was created.

fdateLastAccess Specifies the date the file was last accessed.

ftimeLastAccess Specifies the time the file was last accessed.

fdateLastWrite Specifies the date the file was last written to.

ftimeLastWrite Specifies the time the file was last written to.

cbFile Specifies the end of file data.

cbFileAlloc Specifies the allocated file size.

attrFile Specifies the file attributes.

cchName Specifies the length of the null-terminated filename.

achName[13] Specifies the null-terminated filename.

See Also **DosFindFirst, DosFindNext, FDATE, FTIME**

FILELOCK

```
typedef struct _FILELOCK {    /* flock */
    LONG lOffset;
    LONG lRange;
} FILELOCK;
```

The **FILELOCK** structure contains information about the starting position and number of bytes of a portion of a file to be locked or unlocked.

Fields

lFileOffset Specifies the offset from the beginning of the file to the start of the area to be locked or unlocked.

lRangeLength Specifies the length of the locked or unlocked area (in bytes).

See Also **DosFileLocks**

FILESTATUS

```
typedef struct _FILESTATUS {    /* fsts */
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

The **FILESTATUS** structure contains information about the status of a file.

Fields

fdateCreation Specifies the date the file was created.

ftimeCreation Specifies the time the file was created.

fdateLastAccess Specifies the date the file was last accessed.

ftimeLastAccess Specifies the time the file was last accessed.

fdateLastWrite Specifies the date the file was last written to.

ftimeLastWrite Specifies the time the file was last written to.

cbFile Specifies the end of file data.

cbFileAlloc Specifies the allocated file size.

attrFile Specifies the file attributes.

Comments The **cbFile**, **cbFileAlloc**, and **attrFile** fields are not used by the **DosSetFileInfo** function.

See Also **DosQFileInfo, DosSetFileInfo**

■ FONTINFO

```
typedef struct _FONTINFO {    /* finfo */
    USHORT idCodePage;
    USHORT idFont;
} FONTINFO;
```

The **FONTINFO** structure specifies the code-page and font identifiers for a printer font.

Fields

idCodePage Specifies the code-page ID. It can be one of the following values:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic

idFont Specifies the font. The permitted font ID depends on the printer and on the loaded fonts.

See Also

PRT_ACTIVATEFONT, PRT_QUERYACTIVEFONT, PRT_VERIFYFONT

■ FRAME

```
typedef struct _FRAME {    /* frm */
    BYTE bCharsPerLine;
    BYTE bLinesPerInch;
} FRAME;
```

The **FRAME** structure contains frame-control information for a printer.

Fields

bCharsPerLine Specifies the number of characters on a line, either 80 or 132.

bLinesPerInch Specifies the number of lines per inch, either 6 or 8.

See Also

PRT_GETFRAMECTL, PRT_SETFRAMECTL

■ FSALLOCATE

```
typedef struct _FSALLOCATE {    /* fsalloc */
    ULONG  idFileSystem;
    ULONG  cSectorUnit;
    ULONG  cUnit;
    ULONG  cUnitAvail;
    USHORT cbSector;
} FSALLOCATE;
```

The **FSALLOCATE** structure contains information about a disk drive.

Fields

idFileSystem Specifies the file-system identifier.

cSectorUnit Specifies the number of sectors per allocation unit.

cUnit Specifies the number of allocation units.

cUnitAvail Specifies the available allocation units.

cbSector Specifies the bytes per sector.

See Also **DosQFSInfo**

FSINFO

```
typedef struct _FSINFO {      /* fsinf */
    FDATE fdateCreation;
    FTIME ftimeCreation;
    VOLUMELABEL vol;
} FSINFO;
```

The **FSINFO** structure contains information about the volume label of a disk.

Fields

fdateCreation Specifies the date the volume label was created.

ftimeCreation Specifies the time the volume label was created.

vol Specifies a **VOLUMELABEL** structure that will contain the name of the volume label.

See Also **DosQFSInfo, VOLUMELABEL**

FTIME

```
typedef struct _FTIME {      /* ftime */
    unsigned twosecs : 5;
    unsigned minutes : 6;
    unsigned hours   : 5;
} FTIME;
```

The **FTIME** structure contains the time in seconds, minutes, and hours.

Fields

twosecs Specifies the number of seconds divided by two. To get the actual value, you must multiply it by two. For example, a value of 1 specifies 2 seconds, a value of 2 specifies 4 seconds, and so on.

minutes Specifies the minutes.

hours Specifies the hours.

See Also **FILEFINDBUF, FILESTATUS**

■ GINFOSEG

```
typedef struct _GINFOSEG {    /* gis */
    ULONG   time;
    ULONG   msecs;
    UCHAR   hour;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    USHORT  timezone;
    USHORT  cusecTimerInterval;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    UCHAR   weekday;
    UCHAR   uchMajorVersion;
    UCHAR   uchMinorVersion;
    UCHAR   chRevisionLetter;
    UCHAR   sgCurrent;
    UCHAR   sgMax;
    UCHAR   cHugeShift;
    UCHAR   fProtectModeOnly;
    USHORT  pidForeground;
    UCHAR   fDynamicSched;
    UCHAR   csecMaxWait;
    USHORT  cmsecMinSlice;
    USHORT  cmsecMaxSlice;
    USHORT  bootdrive;
    UCHAR   amecRAS[32];
    UCHAR   csgWindowableVioMax;
    UCHAR   csgPMMax;
} GINFOSEG;
```

The **GINFOSEG** structure contains various global information.

Fields

time Specifies the time from January 1, 1970 (in seconds).

msecs Specifies the current system time (in milliseconds).

hour Specifies the current hour using values from 0 through 23.

minutes Specifies the current minute using values from 0 through 59.

seconds Specifies the current second using values from 0 through 59.

hundredths Specifies the current hundredths of a second using values from 0 through 99.

timezone Specifies the difference (in minutes) between the current time zone and Greenwich Mean Time (GMT). This field is positive for time zones west of Greenwich; it is negative for time zones east of Greenwich. For example, for Eastern Standard Time this field is 300 (that is, five hours, 5×60, after GMT). If this field is -1, the time zone is undefined.

cusecTimerInterval Specifies the timer interval (in milliseconds).

day Specifies the current day of the month using values from 1 through 31.

month Specifies the current month of the year using values from 1 through 12.

year Specifies the current year.

weekday Specifies the current day of the week using values from 0 through 6 (Sunday equals zero).

uchMajorVersion Specifies the major version number.

uchMinorVersion Specifies the minor version number.

chRevisionLetter Specifies the revision letter.

sgCurrent Specifies the current foreground screen group.

sgMax Specifies the maximum number of screen groups.

cHugeShift Specifies the shift count for huge segments.

fProtectModeOnly Specifies the protected-mode-only indicator.

pidForeground Specifies the identifier of the current foreground process.

fDynamicSched Specifies the dynamic variation flag (1 equals enabled).

csecMaxWait Specifies the maximum wait (in seconds).

cmsecMinSlice Specifies the minimum time slice (in milliseconds).

cmsecMaxSlice Specifies the maximum time slice (in milliseconds).

bootdrive Specifies the boot drive.

amecRAS[32] Specifies that each bit corresponds to a system-trace major code from 0x0000 through 0x00FF. The most significant bit (leftmost) of the first byte in the array corresponds to major code 0x0000. If a bit is cleared, the trace is disabled. If a bit is set, the trace is enabled.

csgWindowableVioMax Specifies the maximum number of VIO window-compatible sessions.

csgPMMax Specifies the maximum number of Presentation Manager sessions.

See Also **DosGetInfoSeg, LINFOSEG**

■ HOTKEY

```
typedef struct _HOTKEY {    /* htky */
    USHORT fsHotKey;
    UCHAR  uchScancodeMake;
    UCHAR  uchScancodeBreak;
    USHORT idHotKey;
} HOTKEY;
```

The **HOTKEY** structure contains information for the session-manager hot key.

Fields

fsHotKey Specifies the setting for the session-manager hot key. It can be a combination of the following values:

Value	Meaning
RIGHTSHIFT	Right SHIFT key down.
LEFTSHIFT	Left SHIFT key down.
LEFTCONTROL	Left CONTROL key down.
LEFTALT	Left ALT key down.
RIGHTCONTROL	Right CONTROL key down.
RIGHTALT	Right ALT key down.
SCROLLLOCK	SCROLL LOCK key down.
NUMLOCK	NUMLOCK key down.
CAPSLOCK	CAPSLOCK key down.
SYSREQ	SYSREQ key down.

uchScancodeMake Specifies the scan code of the hot-key "make." If this field is given, the system detects the hot key when the user presses the key that generates this scan code.

uchScancodeBreak Specifies the scan code of the hot-key "break." If this field is given, the system detects the hot key when the user releases the key that generates this scan code.

idHotKey Specifies the session-manager hot-key identifier. It must be a value from 0 through 15.

Comments

The **scancodeMake** and **scancodeBreak** fields are mutually exclusive; only one may be specified.

See Also

KBD_GETSESMGRHOTKEY, KBD_SETSESMGRHOTKEY

KBDINFO

```
typedef struct _KBDINFO {     /* kbst */
    USHORT cb;
    USHORT fsMask;
    USHORT chTurnAround;
    USHORT fsInterim;
    USHORT fsState;
} KBDINFO;
```

The **KBDINFO** structure contains status information for a logical keyboard.

Fields

cb Specifies the length (in bytes) of the **KBDINFO** structure. It must be set to 10.

fsMask Specifies the current keyboard modes. It can be a combination of the following values:

Value	Meaning
KEYBOARD_ECHO_ON	Echo mode turned on.
KEYBOARD_ECHO_OFF	Echo mode turned off.
KEYBOARD_BINARY_MODE	Binary mode turned on.
KEYBOARD_ASCII_MODE	ASCII mode turned on.
KEYBOARD_MODIFY_STATE	The **fsState** field is to be modified. Applies to the **KbdSetStatus** function only.
KEYBOARD_MODIFY_INTERIM	The **fsInterim** field is to be modified. Applies to the **KbdSetStatus** function only.
KEYBOARD_MODIFY_TURNAROUND	The **chTurnAround** field is to be modified. Applies to the **KbdSetStatus** function only.
KEYBOARD_2B_TURNAROUND	Two-byte turn-around character. If not given, the turn-around character is one byte.
KEYBOARD_SHIFT_REPORT	Shift reporting turned on.

Note that echo mode is either turned on or off. Only one input mode, binary or ASCII, can be turned on at any given time.

chTurnAround Specifies the turn-around character. If this field value includes 0x0080, the character is two-bytes packed in the low and high bytes of this field. Otherwise, the character is a single byte in the low byte.

fsInterim Specifies the interim character flags. If this field is 0x0020, the program has requested character conversion. If it is 0x0080, the interim character flag is on.

fsState Specifies the state of the shift keys. It can be any combination of the following values:

Value	Meaning
RIGHTSHIFT	Right SHIFT key down.
LEFTSHIFT	Left SHIFT key down.
CONTROL	CONTROL key down.
ALT	ALT key down.
SCROLLLOCK_ON	SCROLL LOCK mode turned on.
NUMLOCK_ON	NUMLOCK mode turned on.
CAPSLOCK_ON	CAPSLOCK mode turned on.
INSERT_ON	INSERT mode turned on.

See Also

KbdGetStatus, KbdSetStatus

KBDKEYINFO

```
typedef struct _KBDKEYINFO {     /* kbci */
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

The **KBDKEYINFO** structure contains information when a key is pressed.

Fields

chChar Specifies the character derived from translation of the **chScan** field.

chScan Specifies the scan code received from the keyboard, identifying the key pressed. This scan code may be modified during the translation process.

fbStatus Specifies the state of the retrieved scan code. It can be any combination of the following values:

Value	Meaning
SHIFT_KEY_IN	Shift key is received (valid only in binary mode when shift reporting is turned on).
CONVERSION_REQUEST	Conversion requested.
FINAL_CHAR_IN	Final character received.
INTERIM_CHAR_IN	Interim character received.

bNlsShift Specifies a reserved value; must be zero.

fsState Specifies the state of the shift keys. It can be any combination of the following values:

Value	Meaning
RIGHTSHIFT	Right SHIFT key down.
LEFTSHIFT	Left SHIFT key down.
CONTROL	Either CONTROL key down.
ALT	Either ALT key down.
SCROLLLOCK_ON	SCROLL LOCK mode turned on.
NUMLOCK_ON	NUMLOCK mode turned on.
CAPSLOCK_ON	CAPSLOCK mode turned on.
INSERT_ON	INSERT key turned on.
LEFTCONTROL	Left CONTROL key down.
LEFTALT	Left ALT key down.
RIGHTCONTROL	Right CONTROL key down.
RIGHTALT	Right ALT key down.
SCROLLLOCK	SCROLL LOCK key down.
NUMLOCK	NUMLOCK key down.
CAPSLOCK	CAPSLOCK key down.
SYSREQ	SYSREQ key down.

time Specifies the time stamp of the keystroke (in milliseconds).

See Also **KbdCharIn**, **KbdPeek**, KBD_PEEKCHAR

■ KBDTYPE

```
typedef struct _KBDTYPE {    /* kbdtyp */
    USHORT usType;
    USHORT reserved1;
    USHORT reserved2;
} KBDTYPE;
```

The **KBDTYPE** structure contains information about the keyboard type.

Fields

usType Specifies the keyboard type. If this field is 0x0000, an IBM PC/AT keyboard is specified. If it is 0x0001, an IBM enhanced keyboard is specified. Values from 0x0002 to 0x0007 are reserved for Japanese keyboards.

reserved1 Specifies a reserved value; must be zero.

reserved2 Specifies a reserved value; must be zero.

See Also KBD_GETKEYBDTYPE

KBDTRANS

```
typedef struct _KBDTRANS {    /* kbxl */
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
    USHORT fsDD;
    USHORT fsXlate;
    USHORT fsShift;
    USHORT sZero;
} KBDTRANS;
```

The **KBDTRANS** structure contains translated character information.

Fields

chChar Specifies the character value of the translated scan code. The function copies the value to this field before returning.

chScan Specifies the scan code of the keystroke to be translated. This field must be set before the function is called.

fbStatus Specifies the state of the returned scan code. It can be any combination of the following values:

Value	Meaning
SHIFT_KEY_IN	Shift key received (valid only in binary mode when shift reporting is turned on).
CONVERSION_REQUEST	Conversion requested.
FINAL_CHAR_IN	Final character received.
INTERIM_CHAR_IN	Interim character received.

bNlsShift Specifies a reserved value; must be zero.

fsState Specifies the state of the shift keys. It can be one of the following values:

Value	Meaning
RIGHTSHIFT	Right SHIFT key down.
LEFTSHIFT	Left SHIFT key down.
CONTROL	Either CONTROL key down.
ALT	Either ALT key down.
SCROLLLOCK_ON	SCROLL LOCK mode turned on.
NUMLOCK_ON	NUMLOCK mode turned on.
CAPSLOCK_ON	CAPSLOCK mode turned on.
INSERT_ON	INSERT mode turned on.
LEFTCONTROL	Left CONTROL key down.
LEFTALT	Left ALT key down.
RIGHTCONTROL	Right CONTROL key down.
RIGHTALT	Right ALT key down.
SCROLLLOCK	SCROLL LOCK key down.
NUMLOCK	NUMLOCK key down.
CAPSLOCK	CAPSLOCK key down.
SYSREQ	SYSREQ key down.

time Specifies the time stamp of the keystroke (in milliseconds).

fsDD Defined for monitor packets. For more information, see the **DosMonReg** function.

fsXlate Specifies the translation flags. If this field is 0x0000, translation is incomplete. If it is 0x0001, translation is complete.

fsShift Specifies the state of translation across successive calls. Initially, this field should be zero. It should be reset to zero when the caller wants to start a new translation. Note that it may take several calls to the **KbdXlate** function to complete a character, so this field should not be changed unless a new translation is desired. This field is cleared when translation is complete.

sZero Specifies a reserved value; must be zero.

See Also

DosMonReg, KbdXlate

LINECONTROL

```
typedef struct _LINECONTROL {    /* lnctl */
    BYTE bDataBits;
    BYTE bParity;
    BYTE bStopBits;
    BYTE fbTransBreak;
} LINECONTROL;
```

The **LINECONTROL** structure contains line characteristics for a device.

Fields

bDataBits Specifies the number of data bits to be used. It can be one of the following values:

Value	Meaning
0x05	5 data bits
0x06	6 data bits
0x07	7 data bits
0x08	8 data bits

bParity Specifies the type of parity checking. It can be one of the following values:

Value	Meaning
0x00	No parity
0x01	Odd parity
0x02	Even parity
0x03	Mark parity (parity bit always 1)
0x04	Space parity (parity bit always 0)

bStopBits Specifies the number of stop bits used. It can be one of the following values:

Value	Meaning
0x00	1 stop bit
0x01	1.5 stop bits (valid only with 5-bit word length)
0x02	2 stop bits (not valid with 5-bit word length)

fbTransBreak Specifies whether the device is transmitting a break character. If this field is 0x00, a break character is not transmitted. If it is 0x01, a break character is transmitted.

Comments

The ASYNC_GETLINECTRL function (0x0001, 0x0062) uses all four bytes. The ASYNC_SETLINECTRL function (0x0001, 0x0042) uses only the first three bytes.

See Also

ASYNC_GETLINECTRL, ASYNC_SETLINECTRL

■ LINFOSEG

```
typedef struct _LINFOSEG {    /* lis */
    PID     pidCurrent;
    PID     pidParent;
    USHORT  prtyCurrent;
    TID     tidCurrent;
    USHORT  sgCurrent;
    UCHAR   rfProcStatus;
    UCHAR   dummy1;
    BOOL    fForeground;
    UCHAR   typeProcess;
    UCHAR   dummy2;
    SEL     selEnvironment;
    USHORT  offCmdLine;
    USHORT  cbDataSegment;
    USHORT  cbStack;
    USHORT  cbHeap;
    HMODULE hmod;
    SEL     selDS;
} LINFOSEG;
```

The **LINFOSEG** structure contains information local to the current process.

Fields

pidCurrent Specifies the identifier of the current process.

pidParent Specifies the identifier of the parent process.

prtyCurrent Specifies the priority of the current thread.

tidCurrent Specifies the identifier of the current thread.

sgCurrent Specifies the current screen group.

rfProcStatus Specifies the subscreen group.

dummy1 Reserved.

fForeground Specifies that the current process is in foreground.

typeProcess Specifies the process type. It can be one of the following values:

Value	Meaning
0	Process is running in a full-screen protected mode session.
1	Process is running in the compatibility box.

Value	Meaning
2	Process is running in a VIO-windowed session.
3	Process is running in the Presentation Manager screen group.
4	Process is running as a detached process.

dummy2 Reserved.

selEnvironment Specifies the selector to the application's copy of the environment.

offCmdLine Specifies the offset to the environment where the command line that is used to run the current application is copied.

cbDataSegment Specifies the size of the default data segment.

cbStack Specifies the size of the stack.

cbHeap Specifies the size of the heap.

hmod Identifies the program.

selDS Specifies the default data segment.

Comments

The following fields are contained in registers at startup:

Field	Register
SelEnvironment	**ax**
offCmdLine	**bx**
cbDataSegment	**cx**
cbStack	**dx**
cbHeap	**si**
hmod	**di**
selDS	**ds**

See Also

DosGetInfoSeg, GINFOSEG

■ MODEMSTATUS

```
typedef struct _MODEMSTATUS {    /* mdmst */
    BYTE fbModemOn;
    BYTE fbModemOff;
} MODEMSTATUS;
```

The **MODEMSTATUS** structure contains information about modem-control signals.

Fields

fbModemOn Specifies the modem-control signals to be enabled. It can be one or both of the following values:

Value	Meaning
DTR_ON	Data-terminal-ready (DTR) signal enabled.
RTS_ON	Ready-to-transmit (RTS) signal enabled.

If it is 0x00, no signals are enabled.

fbModemOff Specifies the modem-control signals to be disabled. It can be one or both of the following values:

Value	Meaning
DTR_OFF	Data-terminal-ready (RTR) signal disabled.
RTS_OFF	Ready-to-transmit (RTS) signal disabled.

If it is 0xFF, no signals are enabled.

Comments

Any values other than those specified for the **fbModemOn** and **fbModemOff** fields will cause an error value.

See Also

ASYNC_SETMODEMCTRL

■ MONIN

```
typedef struct _MONIN {    /* mnin */
    USHORT cb;
    BYTE abReserved[18];
    BYTE abBuffer[108];
} MONIN;
```

The **MONIN** structure contains monitor-input information.

Fields

cb Specifies the length of the structure (in bytes). The structure must be at least 64 bytes; 128 bytes is the recommended length.

abReserved[18] Specifies a reserved space.

abBuffer[108] Specifies a buffer area which must be greater than or equal to the buffer used by the device driver.

See Also

DosMonReg

■ MONITORPOSITION

```
typedef struct _MONITORPOSITION {    /* mnpos */
    USHORT fPosition;
    USHORT index;
    PBYTE  pbInBuf;
    USHORT offOutBuf;
} MONITORPOSITION;
```

The **MONITORPOSITION** structure contains information about a monitor.

Fields

fposition Specifies the position-flag parameter used in the **DosMonReg** function. It can be one of the following values:

Value	Meaning
MONITOR_DEFAULT	Place the monitor anywhere in the chain.
MONITOR_BEGIN	Place the monitor at the beginning of the chain, in front of any other monitors already in the chain.
MONITOR_END	Place the monitor at the end of the chain, after any other monitors already in the chain.

index Specifies a device-specific value.

pbInBuf Points to the monitor-input buffer that is initialized by the monitor dispatcher and used by the **DosMonRead** function.

offOutBuf Specifies the offset to the monitor-output buffer that is initialized by the monitor dispatcher and used by the **DosMonWrite** function.

See Also **DosMonRead**, **DosMonReg**, **DosMonWrite**, MON_REGISTERMONITOR

■ MONOUT

```
typedef struct _MONOUT {    /* mnout */
    USHORT cb;
    BYTE abReserved[18];
    BYTE abBuffer[108];
} MONOUT;
```

The **MONOUT** structure contains monitor-output information.

Fields

cb Specifies the length of the structure (in bytes). The structure must be at least 64 bytes; 128 bytes is the recommended length.

abReserved[18] Specifies a reserved space.

abBuffer[108] Specifies a buffer area which must be greater than or equal to the buffer used by the device driver.

See Also **DosMonReg**

■ MOUEVENTINFO

```
typedef struct _MOUEVENTINFO {    /* mouev */
    USHORT fs;
    ULONG  time;
    USHORT row;
    USHORT col;
} MOUEVENTINFO;
```

The **MOUEVENTINFO** structure contains information about a mouse event.

Fields

fs Specifies the action that generated the mouse event. It can be any combination of the following values:

Value	Meaning
MOUSE_MOTION	Mouse moved with no buttons down.
MOUSE_MOTION_WITH_BN1_DOWN	Mouse moved with button 1 down.
MOUSE_BN1_DOWN	Button 1 down.

Value	Meaning
MOUSE_MOTION_WITH_BN2_DOWN	Mouse moved with button 2 down.
MOUSE_BN2_DOWN	Button 2 down.
MOUSE_MOTION_WITH_BN3_DOWN	Mouse moved with button 3 down.
MOUSE_BN3_DOWN	Button 3 down.

If the mouse button is released with no motion, this field is zero.

time Specifies the number of milliseconds since MS OS/2 was booted.

row Specifies the *x*-coordinate of the mouse.

col Specifies the *y*-coordinate of the mouse.

See Also **MouReadEventQue**

MOUQUEINFO

```
typedef struct _MOUQUEINFO {    /* mouqi */
    USHORT cEvents;
    USHORT cmaxEvents;
} MOUQUEINFO;
```

The **MOUQUEINFO** structure contains information about the mouse queue.

Fields

cEvents Specifies the number of event-queue elements. It can be any value between zero and the maximum queue size.

cmaxEvents Specifies the maximum queue size (the maximum number of queue elements).

See Also **MouGetNumQueEl**

MUXSEM

```
typedef struct _MUXSEM {    /* mxs */
    USHORT zero;
    HSEM   hsem;
} MUXSEM;
```

The **MUXSEM** structure contains the semaphore used in the **MUXSEMLIST** structure.

Fields

zero Specifies a reserved value; must be zero.

hsem Identifies the semaphore. The handle must have been created previously by using the **DosCreateSem** or **DosOpenSem** function.

See Also **DosCreateSem, DosOpenSem, MUXSEMLIST**

■ MUXSEMLIST

```
typedef struct _MUXSEMLIST {    /* mxsl */
    USHORT cmxs;
    MUXSEM amxs[16];
} MUXSEMLIST;
```

The **MUXSEMLIST** structure contains a list of up to 16 semaphores.

Fields

cmxs Specifies the number of semaphores in the list.

amxs[16] Specifies an array of **MUXSEM** structures.

See Also

DosMuxSemWait, MUXSEM

■ NOPTRRECT

```
typedef struct _NOPTRRECT {    /* mourt */
    USHORT row;
    USHORT col;
    USHORT cRow;
    USHORT cCol;
} NOPTRRECT;
```

The **NOPTRRECT** structure contains the exclusion rectangle for the mouse.

Fields

row Specifies the *x*-coordinate of the upper-left corner.

col Specifies the *y*-coordinate of the upper-left corner.

cRow Specifies the *x*-coordinate of the lower-right corner.

cCol Specifies the *y*-coordinate of the lower-right corner.

Comments

The units for these fields depend on the current video mode. For text mode, values are given in character cells. For graphics mode, values are given in pels. The fields must not exceed the minimum and maximum coordinate values for screen height and width.

See Also

MouRemovePtr

PIDINFO

```
typedef struct _PIDINFO {     /* pidi */
    PID pid;
    TID tid;
    PID pidParent;
} PIDINFO;
```

The **PIDINFO** structure contains process identifiers.

Fields

pid Specifies the process identifier of the calling process.

tid Specifies the thread identifier of the calling thread.

pidParent Specifies the process identifier of the parent process of the calling process.

See Also

DosGetPID

PIPEINFO

```
typedef struct _PIPEINFO {    /* nmpinf */
    USHORT cbOut;
    USHORT cbIn;
    BYTE   cbMaxInst;
    BYTE   cbCurInst;
    BYTE   cbName;
    CHAR   szName[1];
} PIPEINFO;
```

The **PIPEINFO** structure contains named-pipe information retrieved by using the **DosQNmPipeInfo** function. The length of the structure varies depending on the length of the **szName** field.

Fields

cbOut Specifies the size of the buffer for outgoing data.

cbIn Specifies the size of the buffer for incoming data.

cbMaxInst Specifies the maximum number of pipe instances that can be created.

cbCurInst Specifies the number of current pipe instances.

cbName Specifies the length of the pipe name.

szName[1] Contains a null-terminated string with the pipe name, including the computer name if the pipe is remote.

See Also

DosQNmPipeInfo

PTRACEBUF

```
typedef struct _PTRACEBUF {    /* ptrcbf */
    PID    pid;
    TID    tid;
    USHORT cmd;
    USHORT value;
    USHORT offv;
    USHORT segv;
    USHORT mte;
    USHORT rAX;
    USHORT rBX;
    USHORT rCX;
    USHORT rDX;
    USHORT rSI;
    USHORT rDI;
    USHORT rBP;
    USHORT rDS;
    USHORT rES;
    USHORT rIP;
    USHORT rCS;
    USHORT rF;
    USHORT rSP;
    USHORT rSS;
} PTRACEBUF;
```

The **PTRACEBUF** structure contains various debugging information.

Fields

pid Specifies the process identifier of the program being debugged.

tid Specifies the thread identifier of the program being debugged.

cmd Specifies the command to carry out. It can be one of the following values:

Value	Meaning
0x0001	Read memory I-space.
0x0002	Read memory D-space.
0x0003	Read registers.
0x0004	Write memory I-space.
0x0005	Write memory D-space.
0x0006	Write registers.
0x0007	Go (with signal).
0x0008	Terminate child process.
0x0009	Single step.
0x000A	Stop child process.
0x000B	Freeze child process.
0x000C	Resume child process.
0x000D	Convert segment number to selector.
0x000E	Get floating-point registers. The **segv** and **offv** fields must specify the address of a 94-byte buffer that receives the floating-point register values.

Value	Meaning
0x000F	Set floating-point registers. The **segv** and **offv** fields must specify the address of a 94-byte buffer that contains the floating-point register values.
0x0010	Get library-module name. The **value** field must contain the handle of the library module. The **segv** and **offv** fields must contain the address of the buffer that receives the name. This command should be used instead of the **DosGetModHandle** and **DosGetModName** functions to verify the name of a library loaded by the program being debugged.

When the function returns, it copies a code that specifies the command result to the field. The return code can be one of the following values:

Value	Meaning
0x0000	Success return code.
0xFFFF	Error. The error code is in the **value** field.
0xFFFE	About to receive signal.
0xFFFD	Single-step interrupt.
0xFFFC	Hit break point.
0xFFFB	Parity error.
0xFFFA	Process dying.
0xFFF9	General protection fault occurred. The fault type is in the **value** field. The **segv** and **offv** fields contain the address that caused the fault.
0xFFF8	Library module has just been loaded. The **value** field contains the library-module handle.
0xFFF7	Process has not used 287 yet.

value Specifies the value to be used for a given command, or a return value from a command. If an error occurs, the field is set to one of the following values:

Value	Meaning
0x0001	Bad command.
0x0002	Child process not found.
0x0005	Child process untraceable.

offv Specifies the offset from the given segment.

segv Specifies a segment selector.

mte Specifies the module handle that contains the segment.

rAX Specifies the **ax** register.

rBX Specifies the **bx** register.

rCX Specifies the **cx** register.

rDX Specifies the **dx** register.

rSI Specifies the **si** register.

rDI Specifies the **di** register.

rBP Specifies the **bp** register.

rDS Specifies the **ds** register.

rES Specifies the **es** register.

rIP Specifies the **ip** register.

rCS Specifies the **cs** register.

rF Specifies flags.

rSP Specifies the **sp** register.

rSS Specifies the **ss** register.

See Also **DosGetModHandle, DosGetModName, DosPTrace**

PTRDRAWFUNCTION

```
typedef struct _PTRDRAWFUNCTION {    /* ptrdfnc */
    PFN pfnDraw;
    PCH pchDataSeg;
} PTRDRAWFUNCTION;
```

The **PTRDRAWFUNCTION** structure contains information about a pointer-draw function.

Fields

pfnDraw Points to the pointer-draw function.

pchDataSeg Points to the data segment of the pointer-draw function.

See Also MOU_SETPROTDRAWADDRESS, MOU_SETREALDRAWADDRESS, PTR_GETPTRDRAWADDRESSFUNCTION

PTRLOC

```
typedef struct _PTRLOC {    /* moupl */
    USHORT row;
    USHORT col;
} PTRLOC;
```

The **PTRLOC** structure contains the position of the mouse.

Fields

row Specifies the *x*-coordinate of the mouse.

col Specifies the *y*-coordinate of the mouse.

Comments The values of the **row** and **col** fields depend on the current video mode of the screen (as defined by the **VioSetMode** function). For text mode, values are given in character cells. For graphics mode, values are given in pels.

See Also **MouGetPtrPos, MouSetPtrPos, VioSetMode**

PTRSHAPE

```
typedef struct _PTRSHAPE {      /* moups */
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

The **PTRSHAPE** structure contains information about the shape of the mouse.

Fields

cb Specifies the length in bytes of the AND and XOR masks.

col Specifies the width of each mask. For text mode, the width is given in character cells. For graphics mode, the width is given in pels. This value must be greater than or equal to 1.

row Specifies the height of each mask. For text mode, the width is given in character cells. For graphics mode, the height is given in pels. This value must be greater than or equal to 1.

colHot Specifies the horizontal offset from the upper-left corner of the pointer shape to the hot spot. For text mode, the offset is given in character cells. For graphics mode, the offset is given in pels.

rowHot Specifies the vertical offset from the upper-left corner of the pointer shape to the hot spot. For text mode, the offset is given in character cells. For graphics mode, the offset is given in pels.

Comments

The **cb** field of this structure is always equal to the height times the width (**row** × **col**). If the current video mode requires multiple bit planes, the **row** and **col** fields specify the width and height of the first plane only, but the function copies all bit planes to the specified buffer.

See Also

MouGetPtrShape, MouSetPtrShape

QUEUERESULT

```
typedef struct _QUEUERESULT {      /* qresc */
    PID pidProcess;
    USHORT usEventCode;
} QUEUERESULT;
```

The **QUEUERESULT** structure contains the result of a queue-reading operation.

Fields

pidProcess Specifies the process identifier of the process that added the element to the queue.

usEventCode Specifies a program-supplied event code. MS OS/2 does not use this field and reserves it for any use a program may make of it.

See Also

DosPeekQueue, DosReadQueue

RATEDELAY

```
typedef struct _RATEDELAY {    /* rtdly */
    USHORT usDelay;
    USHORT usRate;
} RATEDELAY;
```

The **RATEDELAY** structure contains typamatic information.

Fields

usDelay Specifies the typamatic delay (in milliseconds). A value greater than the maximum value defaults to the maximum value.

usRate Specifies the typamatic rate (characters-per-second). A value greater than the maximum value defaults to the maximum value.

See Also

KBD_SETTYPAMATICRATE

RESULTCODES

```
typedef struct _RESULTCODES {    /* resc */
    USHORT codeTerminate;
    USHORT codeResult;
} RESULTCODES;
```

The **RESULTCODES** structure contains the results of a process when it terminates.

Fields

codeTerminate Specifies the child-process identifier if the child process is asynchronous. Otherwise, it specifies the termination code of the child process. The termination code can be one of the following values:

Value	Meaning
TC_EXIT	Normal exit
TC_HARDERROR	Hard-error termination
TC_TRAP	Trap operation
TC_KILLPROCESS	Unintercepted **DosKillProcess** function

codeResult Specifies the result code of the terminating process in its last call to the **DosExit** function. Specifies the exit code of the child process if the child process is synchronous. This field is not used for an asynchronous child process. The exit code is specified in the last call by the child process to the **DosExit** function.

See Also

DosCwait, DosExecPgm, DosExit, DosKillProcess

RXQUEUE

```
typedef struct _RXQUEUE {    /* rxq */
    USHORT cch;
    USHORT cb;
} RXQUEUE;
```

The **RXQUEUE** structure contains the number of characters in the queue and the size of the queue.

Fields

cch Specifies the number of characters received or to be transmitted in the device-driver queue.

cb Specifies the size of the queue (in bytes).

See Also

ASYNC_GETINQUECOUNT, ASYNC_GETOUTQUECOUNT

SCALEFACT

```
typedef struct _SCALEFACT {    /* mousc */
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

The **SCALEFACT** structure contains information for scaling the mouse.

Fields

rowScale Specifies the vertical scaling factor (the number of mickeys the mouse must move to change the vertical mouse position by one screen unit).

colScale Specifies the horizontal scaling factor (the number of mickeys the mouse must move to change the horizontal mouse position by one screen unit).

Comments

The **rowScale** and **colScale** fields specify mickeys and will always be in the range 1 through 32,767. The screen units may be character cells or pels, depending on the current video mode.

See Also

MouGetScaleFact, MouSetScaleFact

SCREENGROUP

```
typedef struct _SCREENGROUP {    /* scrgrp */
    USHORT idScreenGrp;
    USHORT fTerminate;
} SCREENGROUP;
```

The **SCREENGROUP** structure contains information about the screen group.

Fields

idScreenGrp Specifies the screen-group identifier of the new foreground screen or for notification action. The identifier can range from zero to the maximum number of screen groups. The **sgMax** field in the global descriptor table (GDT) information segment specifies the maximum number of screen groups.

fTerminate Specifies if the screen group is terminating. If it is 0x0000, the screen group is switching. If it is 0xFFFF, the screen group is terminating.

See Also

KBD_SETFGNDSCREENGRP, MOU_SCREENSWITCH

SHIFTSTATE

```
typedef struct _SHIFTSTATE {    /* shftst */
    USHORT fsState;
    BYTE   fNLS;
} SHIFTSTATE;
```

The **SHIFTSTATE** structure contains information about the shift state of the default keyboard of the current screen group.

Fields

fsState Specifies the state of the shift keys. It can be any combination of the following values:

Value	Meaning
RIGHTSHIFT	Right SHIFT key down.
LEFTSHIFT	Left SHIFT key down.
CONTROL	Either CONTROL key down.
ALT	Either ALT key down.
SCROLLLOCK_ON	SCROLL LOCK mode turned on.
NUMLOCK_ON	NUMLOCK mode turned on.
CAPSLOCK_ON	CAPSLOCK mode turned on.
INSERT_ON	INSERT mode turned on.
LEFTCONTROL	Left CONTROL key down.
LEFTALT	Left ALT key down.
RIGHTCONTROL	Right CONTROL key down.
RIGHTALT	Right ALT key down.
SCROLLLOCK	SCROLL LOCK key down.
NUMLOCK	NUMLOCK key down.
CAPSLOCK	CAPSLOCK key down.
SYSREQ	SYSREQ key down.

fNLS Specifies the state of the national-language-support keys. This is zero for the United States.

See Also

KBD_GETSHIFTSTATE, KBD_SETSHIFTSTATE

■ STARTDATA

```
typedef struct _STARTDATA {    /* stdata */
    USHORT Length;
    USHORT Related;
    USHORT FgBg;
    USHORT TraceOpt;
    PSZ    PgmTitle;
    PSZ    PgmName;
    PBYTE  PgmInputs;
    PBYTE  TermQ;
    PBYTE  Environment;
    USHORT InheritOpt;
    USHORT SessionType;
    PSZ    IconFile;
    ULONG  PgmHandle;
    USHORT PgmControl;
    USHORT InitXPos;
    USHORT InitYPos;
    USHORT InitXSize;
    USHORT InitYSize;
} STARTDATA;
```

The **STARTDATA** structure contains information about a session that will be started with the **DosStartSession** function.

Fields

Length Specifies the length of the structure (in bytes). It must be set to 50 bytes.

Related Specifies whether the session created is related to the calling session. If this field is FALSE, the new session is an independent session (not related). If it is TRUE, the new session is a child session (related).

FgBg Specifies whether the new session is started in the foreground or in the background. If this field is TRUE, the session is started in the background. If it is FALSE, the session is started in the foreground.

TraceOpt Specifies whether the program started in the new session is executed under conditions for tracing. If this field is 0, there is no tracing. If it is 1, there is tracing.

PgmTitle Points to the null-terminated string that specifies the program title. The string can be up to 32 bytes long, including the null terminating character. If the address specified is zero or if the null-terminated string is NULL, the initial title is the value of the **PgmName** field minus any leading drive and path information.

PgmName Points to the null-terminated string that specifies the drive, path, and filename of the program to be loaded.

PgmInputs Points to the null-terminated string that specifies the input arguments to be passed to the program.

TermQ Points to the null-terminated string that specifies the full path name of an MS OS/2 queue or is equal to zero. This parameter is optional.

Environment Points to an environment string that is to be passed to the program started in the new session. If this field is zero, the program in the new session inherits the environment of the parent session if the **InheritOpt** field is zero, or the environment of the program calling **DosStartSession** if the **InheritOpt** field is one.

InheritOpt Specifies whether the program started in the new session inherits the environment and open file handles of the calling process. If this field is zero, inheritance is from the parent session. If this field is 1, inheritance is from the calling process.

SessionType Specifies the type of session that should be created. It is one of the following values:

Value	Meaning
0	Use the data specified by the **PgmHandle** field or allow MS OS/2 to establish the session type.
1	Start the process in a full-screen session.
2	Start the process in a window session for programs using the base video subsystem.
3	Start the process in a window session for programs using the Presentation Manager application programming interface.

IconFile Points to a null-terminated string that contains the fully-qualified device, path name, and filename of an icon definition. The system provides an icon for window applications if an icon filename is not provided by the **DosStartSession** call.

PgmHandle Specifies a program handle.

PgmControl Specifies the initial state for a window application. This field is ignored by full-screen sessions. It can be any combination of the following values:

Value	Meaning
0	Invisible
2	Maximize
4	Minimize
8	No auto close
32768	Use specified position and size

InitXPos Specifies the initial *x* coordinate (in pels) for the initial-session window, where (0,0) is the lower-left corner of the display. This field is ignored for full-screen sessions.

InitYPos Specifies the initial *y* coordinate (in pels) for the initial-session window, where (0,0) is the lower-left corner of the display. This field is ignored for full-screen sessions.

InitXSize Specifies the width (in pels) for the initial-session window. This field is ignored for full-screen sessions.

InitYSize Specifies the height (in pels) for the initial-session window. This field is ignored for full-screen sessions.

See Also

DosStartSession

STATUSDATA

```
typedef struct _STATUSDATA {    /* stsdata */
    USHORT Length;
    USHORT SelectInd;
    USHORT BindInd;
} STATUSDATA;
```

The **STATUSDATA** structure contains status information about a session.

Fields

Length Specifies the length of the data structure (in bytes).

SelectInd Specifies whether the target session should be set as selectable or nonselectable. It can be one of the following values:

Value	Meaning
TARGET_UNCHANGED	Leave current setting unchanged.
TARGET_SELECTABLE	Set as selectable.
TARGET_NOT_SELECTABLE	Set as nonselectable.

BindInd Specifies which session to bring to the foreground the next time the parent session is selected. It can be one of the following values:

Value	Meaning
BIND_UNCHANGED	Leave current setting unchanged.
BIND_CHILD	A bond between the parent session and the child session is established. The child session is brought to the foreground the next time the

Value	Meaning
	parent session is selected. If the child session is selected, the child session is brought to the foreground.
BIND_NONE	Any bond previously established with the specified child session is broken. The parent session is brought to the foreground the next time the parent session is selected and the child session is brought to the foreground the next time the child session is selected.

See Also **DosSetSession**

STRINGINBUF

```
typedef struct _STRINGINBUF {     /* kbsi */
    USHORT cb;
    USHORT cchIn;
} STRINGINBUF;
```

The **STRINGINBUF** structure contains information about the length of the buffer used by the **KbdStringIn** function.

Fields

cb Specifies the length of the buffer (in bytes). The maximum value is 0x00FF.

cchIn Specifies the number of bytes read. The maximum value is 0x00FF.

See Also **KbdStringIn**

TRACKFORMAT

```
typedef struct _TRACKFORMAT {     /* trckfmt */
    BYTE bCommand;
    USHORT usHead;
    USHORT usCylinder;
    USHORT usReserved;
    USHORT cSectors;
    struct {
        BYTE bCylinder;
        BYTE bHead;
        BYTE idSector;
        BYTE bBytesSector;
    } FormatTable[1];
} TRACKFORMAT;
```

The **TRACKFORMAT** structure contains information about the disk drive.

Fields

bCommand Specifies the type of track layout. If this field is 0x0000, the track layout contains nonconsecutive sectors or does not start with sector 1. If it is 0x0001, the track layout starts with sector 1 and contains only consecutive sectors.

usHead Specifies the number of the physical head on which to perform the operation.

usCylinder Specifies the cylinder number for the operation.

cSectors Specifies the number of sectors on the track being formatted.

FormatTable[1] Specifies the format table. It is an array of structures that contain the cylinder number, head number, sector identifier, and bytes per sector for each sector on the track. The **bCylinder** field specifies the cylinder number. The **bHead** field specifies the head number. The **idSector** field specifies the sector identifier, and the **bBytesSector** field specifies the number of bytes per sector. The first element defines these values for the first sector. The number of elements depends on the number of sectors on the track. The **bBytesSector** field can be one of the following values:

Value	Meaning
0x0000	128 bytes per sector
0x0001	256 bytes per sector
0x0002	512 bytes per sector
0x0003	1024 bytes per sector

All the cylinder and head numbers must be the same.

See Also DSK_FORMATVERIFY

■ TRACKLAYOUT

```
typedef struct _TRACKLAYOUT {    /* trckl */
    BYTE   bCommand;
    USHORT usHead;
    USHORT usCylinder;
    USHORT usFirstSector;
    USHORT cSectors;
    struct {
        USHORT usSectorNumber;
        USHORT usSectorSize;
    } TrackTable[1];
} TRACKLAYOUT;
```

The **TRACKLAYOUT** structure contains track-layout information.

Fields

bCommand Specifies the type of track layout. If this field is 0x0000, the track layout contains nonconsecutive sectors or does not start with sector 1. If it is 0x0001, the track layout starts with sector 1, and contains only consecutive sectors.

usHead Specifies the physical head on the disk drive on which to perform the operation.

usCylinder Specifies the cylinder number on which to perform the operation.

usFirstSector Specifies the logical sector number at which to start the operation. The logical sector number is the index in the track-layout table to the first sector. Index 0 specifies the first sector, index 1 the second, and so on.

cSectors Specifies the number of sectors on which to perform the operation, up to the maximum specified in the track-layout table. The function does not step heads and tracks.

TrackTable[1] Specifies the track-layout table. It is an array of structures that contain the numbers and sizes of the sectors in the track. The first element in this field defines the sector number and size (in bytes) of the first sector on the track, the second element defines the second sector, and so on. For each element of **TrackTable**, the **usSectorNumber** field specifies the sector number, and the **usSectorSize** field specifies the size of the sector. The number of elements depends on the number of sectors on the track.

See Also

PDSK_READPHYSTRACK, DSK_READTRACK, PDSK_VERIFYPHYSTRACK, DSK_VERIFYTRACK, DSK_WRITETRACK, PDSK_WRITEPHYSTRACK

VIOCONFIGINFO

```
typedef struct _VIOCONFIGINFO {    /* vioin */
    USHORT cb;
    USHORT adapter;
    USHORT display;
    ULONG  cbMemory;
} VIOCONFIGINFO;
```

The **VIOCONFIGINFO** structure contains configuration information about the screen.

Fields

cb Specifies the length of the structure (in bytes). This field must be set to 10 before calling the **VioGetConfig** function.

adapter Specifies the display-adapter type. It can be one of the following values:

Value	Meaning
DISPLAY_MONOCHROME	Monochrome/printer adapter
DISPLAY_CGA	Color graphics adapter
DISPLAY_EGA	Enhanced graphics adapter
DISPLAY_VGA	Video graphics array or IBM Personal System/2 display adapter
DISPLAY_8514A	PS/2 Display adapter 8514/A

display Specifies the display/monitor type. It can be one of the following values:

Value	Meaning
MONITOR_MONOCHROME	Monochrome display
MONITOR_COLOR	Color display
MONITOR_ENHANCED	Enhanced color display
MONITOR_8503	8503 monochrome display
MONITOR_851X_COLOR	8512 or 8513 color display
MONITOR_8514	8514 color display

cbMemory Specifies the amount of memory on the adapter (in bytes).

See Also

VioGetConfig

■ VIOCURSORINFO

```
typedef struct _VIOCURSORINFO {    /* vioci */
    USHORT yStart;
    USHORT cEnd;
    USHORT cx;
    USHORT attr;
} VIOCURSORINFO;
```

The **VIOCURSORINFO** structure contains information about the cursor.

Fields

yStart Specifies the horizontal scan line that marks the top line of the cursor. Scan lines are numbered from 0 to *n*–1, where *n* is the maximum height of a character cell. Scan line 0 is at the top of the character cell.

cEnd Specifies the horizontal scan line that marks the bottom line of the cursor.

cx Specifies the width of the cursor in columns (for text mode) or in pels (for graphics mode). The maximum width in text mode is 1. If zero is given, the function uses a default width: 1 for text mode or the width of a character cell for graphics mode.

attr Specifies the attribute of the cursor. If this field is 0xFFFF, the function hides the cursor (removes it from the screen). Any other value sets the current character attribute of the cursor.

See Also

VioGetCurType, VioSetCurType

■ VIOFONTINFO

```
typedef struct _VIOFONTINFO {    /* viofi */
    USHORT cb;
    USHORT type;
    USHORT cxCell;
    USHORT cyCell;
    PVOID  pbData;
    USHORT cbData;
} VIOFONTINFO;
```

The **VIOFONTINFO** structure contains information about the font.

Fields

cb Specifies the length of the structure (in bytes). It must be set to 14.

type Specifies the request type. This field must be VGFI_GETCURFONT to retrieve the current font. It must be VGFI_GETROMFONT to retrieve a ROM font. It must be 0x0000 to set a font.

cxCell Specifies the width (in pels) of each character cell in the font.

cyCell Specifies the height (in pels) of each character cell in the font.

pbData Points to the buffer that receives the requested font table or can be set to NULL to direct the **VioGetFont** function to supply an address. In the latter case, the function copies the address of the font to this field. The address specifies either a RAM or a ROM address, depending on the request type.

For the **VioSetFont** function, it points to the buffer that contains the font table to set a font. The format of the font table depends on the display adapter and screen mode.

cbData Specifies the length of the font (in bytes).

Comments When requesting a ROM font, the **cxCell** and **cyCell** fields must be set before calling the **VioGetFont** function. These fields identify the font to be retrieved.

See Also **VioGetFont, VioSetFont**

VIOINTENSITY

```
typedef struct _VIOINTENSITY {    /* vioint */
    USHORT  cb;
    USHORT  type;
    USHORT  fs;
} VIOINTENSITY;
```

The **VIOINTENSITY** structure contains status information about foreground and background color.

Fields

cb Specifies the length of the structure (in bytes). It must be set to 6.

type Specifies the request type. To retrieve the blink/background intensity switch, this field must be set to 0x0002.

fs Specifies foreground and background color status. This field must be set to 0x0000 for blinking foreground colors, or 0x0001 for high-intensity background colors.

See Also **VioGetState, VioSetState, VIOOVERSCAN, VIOPALSTATE**

VIOMODEINFO

```
typedef struct _VIOMODEINFO {    /* viomi */
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
} VIOMODEINFO;
```

The **VIOMODEINFO** structure contains information about the screen mode.

Fields

cb Specifies the length of the data structure (in bytes). This field must be set to 12.

fbType Specifies the screen mode. It is one of the following values:

Value	Meaning
VGMT_OTHER	Set adapter to other than a monochrome/printer adapter. If this value is not given, the monochrome/printer adapter is assumed by default.
VGMT_GRAPHICS	Set graphics mode. If this value is not given, the adapter is set to text mode.
VGMT_DISABLEBURST	Disable color-burst mode. If this value is not given, color-burst mode is enabled.

color Specifies the number of colors (defined as a power of 2). This is equivalent to the number of color bits that define the color. It is one of the following values:

Value	Meaning
COLORS_2	2 colors
COLORS_4	4 colors
COLORS_16	16 colors

col Specifies the number of text columns.

row Specifies the number of text rows.

hres Specifies the number of pel columns (horizontal resolution).

vres Specifies the number of pel rows (vertical resolution).

See Also **VioGetMode, VioSetMode**

VIOOVERSCAN

```
typedef struct _VIOOVERSCAN {    /* vioos */
    USHORT  cb;
    USHORT  type;
    USHORT  color;
} VIOOVERSCAN;
```

The **VIOOVERSCAN** structure contains the overscan (border) screen color.

Fields

cb Specifies the length of the structure (in bytes). It must be set to 6.

type Specifies the request type. To retrieve the overscan (border) color, this field must be set to 0x0001.

color Specifies the color value.

See Also **VioGetState, VioSetState, VIOINTENSITY, VIOPALSTATE**

VIOPALSTATE

```
typedef struct _VIOPALSTATE {    /* viopal */
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;
```

The **VIOPALSTATE** structure contains the screen-palette registers.

Fields

cb Specifies the length of the structure (in bytes). The length determines how many palette registers are retrieved. The maximum length is 38 bytes for 16 registers.

type Specifies the request type. To retrieve the palette register state, this field must be set to 0x0000.

iFirst Specifies the first palette register to be retrieved. This field must be a value from 0x0000 to 0x000F. The function retrieves the palette registers in sequential order. The number of registers retrieved depends on the structure size specified by the **cb** field.

acolor[1] Specifies the array that receives the color values for the palette registers.

See Also **VioGetState, VioSetState, VIOINTENSITY, VIOOVERSCAN**

VIOPHYSBUF

```
typedef struct _VIOPHYSBUF {     /* viopb */
    PBYTE pBuf;
    ULONG cb;
    SEL   asel[1];
} VIOPHYSBUF;
```

The **VIOPHYSBUF** structure contains information about the physical video buffer.

Fields

pBuf Points to the physical video buffer. The address must be in the range 0x000A0000 through 0x000BFFFF; this depends on the display adapter and the video mode.

cb Specifies the length of the physical video buffer (in bytes).

asel[1] Specifies the array that receives the selectors used to address the physical video buffer. If more than one selector is received, the first selector addresses the first 64K bytes of the physical video buffer, the second selector addresses the next 64K bytes, and so on. The number of selectors depends on the actual size of the physical buffer as specified by the **cb** field. The last selector may address less than 64K bytes of buffer.

Comments The actual size of the **asel[1]** field depends on the size of physical memory. The program must ensure that there is adequate space to receive all selectors.

See Also **VioGetPhysBuf**

VOLUMELABEL

```
typedef struct _VOLUMELABEL {    /* vol */
    BYTE cch;
    CHAR szVolLabel[12];
} VOLUMELABEL;
```

The **VOLUMELABEL** structure contains the volume label.

Fields

cch Specifies the length of the **achVolLabel[12]** field (excluding the null-terminating character).

achVolLabel[12] Specifies a null-terminated string that specifies the volume label. When a volume label is being set by using the **DosSetFSInfo** function, any trailing spaces are ignored.

See Also **DosQFSInfo, DosSetFSInfo**

Chapter 5

File Formats

Chapter 5

5.1 Introduction

This chapter describes the format of the files and related structures used by MS OS/2 functions. The following topics are described in detail:

- Keyboard translation tables
- Video fonts
- Resources

This chapter describes the formats as returned by or required by the MS OS/2 functions that use them. The formats described here may not fully describe the format of data when it is stored in an MS OS/2 system file. For example, the system default keyboard translation tables are stored in the *keyboard.dcp* file. This file usually contains header information and several translation tables. Although the translation-table format is described in this chapter, the header information and the organization of the tables in the files are not.

In general, this chapter describes only the details needed to develop data formats for use with MS OS/2 programs. The programmer can choose an appropriate file-storage format.

5.2 Keyboard Translation Tables

This section describes the format and contents of MS OS/2 translation tables. MS OS/2 uses translation tables to translate keystroke scan codes into character values.

5.2.1 Predefined Translation Tables

MS OS/2 provides several predefined translation tables. These tables, defined in the *keyboard.dcp* file, specify the translations for keyboard scan codes to character values for a variety of character sets and languages. Each translation table is identified by a code-page identifier. The code-page ID may be used in the **DosSetCp**, **KbdSetCp**, and **VioSetCp** functions to set the translation table for the system. The **DosGetCp**, **KbdGetCp**, and **VioGetCp** functions also retrieve the code-page ID for the current system translation table.

The following is a list of the MS OS/2 predefined translation tables and their code-page identifiers:

Number	Code page
437	United States
850	Multilingual
860	Portuguese
863	French-Canadian
865	Nordic
0x0000	Default (none)

A user can set the translation tables for the system by using the **codepage** and **devinfo** commands in the *config.sys* file. The **keyb** command can be used to change the current translation table.

5.2.2 Translation-Table Format

MS OS/2 lets a program create and set custom translation tables for the keyboard by using the **KbdSetCustXt** function. The function takes a pointer to translation table. The translation table is a structure that has the following general form:

Translation-table header
Key-definition 1
Key-definition 2
.
.
.
Key-definition 127
Accent-key table

The translation-table header defines the translation table's code-page ID, the size of the translation table, the keyboard for which it was designed, and other information about the translation table. The key-definition entries define key-translation type, the accent keys that can be used in combination with this key, and the actual translated character values. A translation table may have up to 127 key-definition entries. The accent-table entry defines the scan- and character-code translations for accent-and-character key combinations. This accent table contains seven accent entries and accent-key definitions.

```
struct {
    USHORT XTableID;
    USHORT XTableFlags1;
    USHORT XTableFlags2;
    USHORT KbdType;
    USHORT KbdSubType;
    USHORT XTableLen;
    USHORT EntryCount;
    USHORT EntryWidth;
    USHORT Country;
    USHORT TableTypeID;
    USHORT Reserved[10];
    struct {
            USHORT AccentFlags:7;
            USHORT KeyType:9;
            CHAR Char1;
            CHAR Char2;
            CHAR Char3;
            CHAR Char4;
            CHAR Char5;
    } KeyDef[127];
    struct {
            BYTE NonAccent[2];
            BYTE CtlAccent[2];
            BYTE AltAccent[2];
            BYTE Map[20][2];
    } AccentEntry[7];
};
```

Field	Description
XTableID	Specifies the code-page ID for this translation.
XTableFlags1	Specifies the first set of table flags. For more information, see the values listed in Table 5.1.
XTableFlags2	Specifies a reserved value; must be zero.
KbdType	Specifies the keyboard type. This field is 0x0000 for an IBM PC/AT keyboard and 0x0001 for an IBM Enhanced keyboard.
KbdSubType	Specifies a reserved value; must be zero.
XTableLen	Specifies the length of the translation table (in bytes).
EntryCount	Specifies the number of key-definition entries.
EntryWidth	Specifies the width of each key-definition entry (in bytes).
Country	Specifies the country-code or language ID. This ID consists of two letters that represent the name of a country. The first letter is stored in the high-order byte, the second in the low-order byte. For more information, see the codes listed in Table 5.2.
TableTypeID	Specifies the table type. The low-order byte specifies the type, the high-order byte the sub-type. This field must be 0x0001.
Reserved[10]	Specifies an array of reserved values. Each element must be zero.
AccentFlags	Specifies the translation for accent keys. This field occupies bits 0 through 6.
KeyType	Specifies the translation of the keys. This field occupies bits 11 through 15.
Char1	Specifies a translated-character value. Typically used when no shift keys are pressed.
Char2	Specifies a translated-character value. Typically used when shift keys are pressed.
Char3	Specifies a translated-character value. Typically used when the ALT GR (alternate-graphics) key is pressed.
Char4	Specifies a translated-character value.
Char5	Specifies a translated-character value.

Field	Description
NonAccent[2]	Specifies the character value and scan code for the key when not used as an accent character. The first byte contains the character value, the second the scan code.
CtlAccent[2]	Specifies the character value and scan code for the key when used with the CONTROL key. The first byte contains the character value, the second the scan code.
AltAccent[2]	Specifies the character value and scan code for the key when used with the ALT key. The first byte contains the character value, the second the scan code.
Map[20][2]	Specifies an array of scan-code and character-value pairs for accented translation. The array has 20 elements. Each element has two bytes; the first byte contains the scan code of a key to be accented and the second contains the character value of the accented key.

The **XTableFlags1** field can be any combination of the values listed in Table 5.1:

Table 5.1 Table-Flag Values

Value	Meaning
0x0001	SHIFT+ALT is used in place of CONTROL+ALT.
0x0002	Left ALT key is the ALT GR (alternate-graphics) key.
0x0004	Right ALT key is the ALT GR (alternate-graphics) key.
0x0008	CAPSLOCK key is interpreted as a SHIFTLOCK key.
0x0010	Default table for the language. Used by the **keyb** command to locate the default translation table if switching between several translation tables.
0x0020	SHIFTLOCK key is a toggle key. If not given, the key is a latch key.
0x0040	Accent is sent as a character. If not valid, beep is sounded.
0x0080	When the CAPSLOCK is down and the SHIFT key is pressed, the **Char5** field is used in the key-definition entry.

The **Country** field specifies the country or language identifier. It can be any of the codes listed in Table 5.2:

Table 5.2 Country and Language Codes

Code	Country/Language
US	United States
UK	United Kingdom
GR	Germany
FR	France
IT	Italy
SP	Spain
DK	Denmark
NL	Netherlands
SU	Finland
NO	Norway
PO	Portugal
SV	Sweden
SF	Switzerland (French)
SG	Switzerland (German)
CF	French-Canadian
BE	Belgium
LA	Latin America (Spanish)

Note that each accent entry should have the space character defined as one of its accented characters and be translated to the same value as the accent character itself. The reason for this is that, by definition, an accent key followed by the space character maps to the accent character alone. If the table is not set up this way, a "not-an-accent" beep sounds when the accent key, followed by a space, is pressed.

5.2.3 Key Types

The **KeyType** field specifies whether the scan code represents an alphabetic, special, function, shift, or other type of key. It also defines how to translate the key when a given shift key is down or active. This field can be one of the following values:

Value	Meaning
0x0001	Alphabetic-character key
0x0002	Special nonalphabetic-character key
0x0003	Special nonalphabetic-character key with CAPSLOCK translation

Value	Meaning
0x0004	Special nonalphabetic-character key with ALT translation
0x0005	Special nonalphabetic-character key with CAPSLOCK and ALT translations
0x0006	Function key
0x0007	Keypad key
0x0008	Action key that performs a special action when the CONTROL key is pressed
0x0009	PRINTSCREEN key
0x000A	SYSREQ key
0x000B	Accent key (also called a dead key)
0x000C	Shift key (for example, SHIFT or CONTROL)
0x000D	General toggle key
0x000E	ALT key
0x000F	NUMLOCK key
0x0010	CAPSLOCK key
0x0011	SCROLL LOCK key
0x0012	Extended-shift key
0x0013	Extended-toggle key
0x0014	Special character key with CAPSLOCK translations for foreign-language keyboards
0x0015	Special character key with ALT translations for foreign-language keyboards

The **AccentFlags** field of a key-definition entry has seven flags that are individually set if a corresponding entry in the accent table applies to this scan code. If an accent key is pressed immediately before the current key, and if the bit for that accent key is set in the **AccentFlags** field for the current key, the corresponding accent-table entry is searched for the replacement character value. If no replacement is found, the "not-an-accent" beep sounds and the accent character and current character are passed as two separate characters.

The SPACEBAR should have a flag set in its **AccentFlags** field for each possible accent (that is, for each defined accent entry in the accent table).

When no shift keys are pressed, the **Char1** field specifies the translated-character value (except where otherwise noted).

The ALT key, the ALT-GR key, or both, may be present on a keyboard as specified by the **XTableFlags1** field in the translation-table header. In most cases, if the ALT GR key is specified, the **Char3** field specifies the translated-character value when the given key is pressed at the same time as the ALT key.

Any key combination that does not have an explicit definition is assumed to be undefined—for example, pressing the CONTROL key with the 3 key. The system marks the keystroke packet as an undefined translation and passes the packet on to any keyboard monitors. The scan code in the packet remains unchanged but the character value is set to zero. Although the system passes the packet to monitors, it does not copy the undefined translation to the keyboard-input buffer.

The system uses the masks listed in Table 5.3 to set and clear the keyboard shift-status word:

Table 5.3 Shift-Key Masks

Key	Char1	Char2	Char3
SHIFT (right)	0x01	0x00	0x00
SHIFT (left)	0x02	0x00	0x00
CONTROL+SHIFT	0x04	0x01	0x04
ALT+SHIFT	0x08	0x02	0x08
SCROLL LOCK	0x10	0x10	0x10
NUMLOCK	0x20	0x20	0x20
CAPSLOCK	0x40	0x40	0x40
SYSREQ	0x80	0x80	–

The following sections describe the key types in detail.

5.2.3.1 Alphabetic Key (Type 0x0001)

An alphabetic key (type 0x0001) is any character key that represents a letter.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char2**
SHIFT and CAPSLOCK	**Char1**
CONTROL	**Char1** to compute an ASCII control value.
ALT	**Char1** to compute an IBM PC keyboard scan code.
ALT GR	**Char3** if this field is not zero.

If a CONTROL key is pressed, the system subtracts 95 from the **Char1** field to compute an ASCII control value. The final value ranges from 1 through 26.

If an ALT key is pressed, the system uses the **Char1** field as an index to a table of IBM PC keyboard scan codes. The final value is two bytes. The first byte is 0x00. The second byte is the corresponding IBM PC scan code.

5.2.3.2 Special-Character Key (Type 0x0002)

A special-character key (type 0x0002) represents a nonalphabetic character for which there is no CAPSLOCK or ALT translation.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char1**
CONTROL	Computed ASCII control code.
ALT	Undefined translation.
ALT GR	**Char3** if this field is not zero.

If a CONTROL key is pressed, the system uses the scan code of the given key to generate an ASCII control code, as shown in the following list:

Scan code	Control code
0x03	0x00
0x07	0x1E
0x0C	0x1F
0x1A	0x1B
0x1B	0x1D
0x2B	0x1C

Only the scan codes listed generate control codes. A hyphen-character (–) key always generates control code 0x1F, even if the corresponding scan code is not listed. A hyphen-character key is any key whose **Char1** field is 0x2D.

5.2.3.3 Special-Character Key (Type 0x0003)

A special-character key (type 0x0003) represents a nonalphabetic character for which there is a CAPSLOCK translation but no ALT translation.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char2**
SHIFT and CAPSLOCK	**Char1**
CONTROL	Computed ASCII control code.

Shift key	Field used
ALT	Undefined translation.
ALT GR	**Char3** if this field is not zero.

If a CONTROL key is pressed, the system uses the scan code of the given key to generate an ASCII control code, as shown in the following list:

Scan code	Control code
0x03	0x00
0x07	0x1E
0x0C	0x1F
0x1A	0x1B
0x1B	0x1D
0x2B	0x1C

Only the scan codes listed generate control codes. A hyphen-character (–) key always generates control code 0x1F, even if the corresponding scan code is not listed. A hyphen-character key is any key whose **Char1** field is 0x2D.

5.2.3.4 Special-Character Key (Type 0x0004)

A special-character key (type 0x0004) represents a nonalphabetic, non-action key for which there is an ALT translation but no CAPSLOCK translation. Typically, these keys represent numeric and punctuation characters. The SPACEBAR key is also a type 0x0004 key.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char1**
CONTROL	Computed ASCII control code.
ALT	Computed extended ASCII code.
ALT GR	**Char3** if this field is not zero.

If a CONTROL key is pressed, the system uses the scan code of the given key to generate an ASCII control code, as shown in the following list:

Scan code	Control code
0x03	0x00
0x07	0x1E
0x0C	0x1F
0x1A	0x1B

Scan code	Control code
0x1B	0x1D
0x2B	0x1C

Only the scan codes listed generate control codes. A hyphen-character (–) key always generates control code 0x1F, even if the corresponding scan code is not listed. A hyphen-character key is any key whose **Char1** field is 0x2D. Both the ALT+SPACEBAR and CONTROL+SPACEBAR combinations generate the ASCII space character.

If the ALT key is pressed, the system uses the scan code of the given key to generate an extended ASCII code, as shown in the following list:

Scan code	Control code
0x02	0x78
0x03	0x79
0x04	0x7A
0x05	0x7B
0x06	0x7C
0x07	0x7D
0x08	0x7E
0x09	0x7F
0x0A	0x80
0x0B	0x81
0x0C	0x82
0x0D	0x83

The final value is two bytes. The first byte is 0x00 or 0xE0. The second byte is the corresponding extended ASCII code.

5.2.3.5 Special-Character Key (Type 0x0005)

A special-character key (type 0x0005) represents a nonalphabetic character that has both CAPSLOCK and ALT translations.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char2**
SHIFT and CAPSLOCK	**Char1**
CONTROL	Computed ASCII control code.

Shift key	Field used
ALT	Computed extended ASCII code.
ALT GR	**Char3** if this field is not zero.

Only the scan codes listed generate control codes. A hyphen-character (–) key always generates control code 0x1F, even if the corresponding scan code is not listed. A hyphen-character key is any key whose **Char1** field is 0x2D.

If the ALT key is pressed, the system uses the scan code of the given key to generate an extended ASCII code, as shown in the following list:

Scan code	Control code
0x02	0x78
0x03	0x79
0x04	0x7A
0x05	0x7B
0x06	0x7C
0x07	0x7D
0x08	0x7E
0x09	0x7F
0x0A	0x80
0x0B	0x81
0x0C	0x82
0x0D	0x83

The final value is two bytes. The first byte is 0x00 or 0xE0. The second byte is the corresponding extended ASCII code.

5.2.3.6 Function Key (Type 0x0006)

A function key (type 0x0006) represents a non-ASCII key that may be used to direct an action. The system uses the **Char1** field to generate an extended ASCII code for the given key. The **Char1** field should be set to the same value as the key—for example, 1 for the F1 key, 2 for the F2 key, and so on. The system generates the extended ASCII code by adding a value to **Char1**, as shown in the following list:

Shift key	Extended code
None	Adds 0x3A to **Char1**. The F11 and F12 keys are always 0x8B and 0x8C, respectively.
SHIFT	Adds 0x53 to **Char1**. The SHIFT+F11 and SHIFT+F12 keys are always 0x8D and 0x8E, respectively.

Shift key	Extended code
CAPSLOCK	Same as no shift key.
CONTROL	Adds 0x5D to **Char1**. The CONTROL+F11 and CONTROL+F12 keys are always 0x8F and 0x90, respectively.
ALT	Adds 0x67 to **Char1**. The ALT+F11 and ALT+F12 keys are always 0x91 and 0x92, respectively.
ALT GR	**Char3** if this field is not zero.

5.2.3.7 Keypad Key (Type 0x0007)

A keypad key (type 0x0007) represents a keypad character such as a direction or a numeric key.

Shift key	Field used
None	**Char1** used to compute an extended ASCII code.
SHIFT	**Char2**
NUMLOCK	**Char2**
SHIFT and NUMLOCK	Same as no shift key.
CAPSLOCK	Same as no shift key.
CONTROL	Special keypad codes.
ALT	Build a character.
ALT GR	**Char3** if this field is not zero.

The following list shows the required **Char1** values based on the key-top labels:

Key-top label	Char1 value
HOME/7	0x00
UP/8	0x01
PAGE UP/9	0x02
–	0x03
LEFT/4	0x04
5	0x05
RIGHT/6	0x06
+	0x07
END/1	0x08
DOWN/2	0x09

Key-top label	Char1 value
PAGE DOWN/3	0x0A
INS/0	0x0B
DEL/.	0x0C

The **Char2** value should represent the ASCII equivalent of the key-top label. For example, **Char2** for the HOME/7 key should be the ASCII character 7.

When the system generates an extended ASCII code, it creates two bytes. The first byte is 0x00 or 0xE0. The second byte is a scan code equal to the **Char1** field plus 0x47. The plus (+) and minus (–) keypad keys never generate extended ASCII values; they always return the **Char2** field.

If the ALT key is pressed and held down, the system builds a character value by accumulating keystrokes. For each keystroke, the system multiplies the accumulated value by 10, then adds the decimal value of the given key. For example, pressing the HOME/7 key adds 7 to the accumulated value. If the result is greater than 255, the high bits are truncated. If any key other than the numeric keys is pressed, the accumulated value is reset to zero. When the ALT key is released, the accumulated value becomes the character value and the scan code is set to zero.

If the CONTROL key is pressed, the system generates special extended ASCII codes for the keypad keys, as shown in the following list:

Key-top label	Extended code
HOME/7	0x77
UP/8	0x8D
PAGE UP/9	0x84
–	0x8E
LEFT/4	0x73
5	0x8F
RIGHT/6	0x74
+	0x90
END/1	0x75
DOWN/2	0x91
PAGE DOWN/3	0x76
INS/0	0x92
DEL/.	0x93

5.2.3.8 Special-Action Key (Type 0x0008)

A special-action key (type 0x0008) represents an action key that carries out a special action when the CONTROL key is pressed. For example, the ENTER key generates the newline character in combination with the CONTROL key. When pressed alone, it generates the carriage-return character. The special action keys are given in the following list:

Shift key	Field used
None	**Char1**
SHIFT	**Char1**
CAPSLOCK	**Char1**
CONTROL	**Char2**
ALT	Undefined translation.
ALT GR	**Char3** if this field is not zero.

5.2.3.9 PRINTSCREEN Key (Type 0x0009)

The PRINTSCREEN (print-screen) key (type 0x0009) directs the system to copy the screen contents to the printer.

Shift key	Field used
None	**Char1**
SHIFT	Directs the system to print the screen.
CAPSLOCK	**Char1**
CONTROL	Directs the system to echo each screen line to the printer.
ALT	Undefined translation.
ALT GR	**Char3** if this field is not zero.

5.2.3.10 SYSREQ Key (Type 0x000A)

The SYSTEQ (system-request) key (type 0x000A) represents a special shift key. The **Char1** field holds a bit mask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. When the user presses this key, the system sets the shift-status word and clears it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a shift key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

5.2.3.11 Accent Key (Type 0x000B)

An accent key (also called a dead key) (type 0x000B) represents a character that is combined with another character to form a new character. For example, an umlaut key can be combined with the letter *u* to form an umlaut-u character. The **Char1**, **Char2**, and **Char3** fields are indexes into the translation table's accent table. Each field must be a value from 1 through 7.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char1**
CONTROL	**Char1**, but use **CtlAccent** field in accent entry.
ALT	**Char1**, but use **AltAccent** field in accent entry.
ALT GR	**Char3**

When an accent key is pressed with a CONTROL or ALT key, the system retrieves the character value from the **CtlAccent[2]** or **AltAccent[2]** field in the accent-table entry indexed by the **Char1** field. These fields contain the scan and character codes for the key. If the fields are both zero, the key has an undefined translation.

When an accent key is pressed by itself, the system uses the **Char1** field as an index to an accent-table entry. When an accent key is pressed with a SHIFT key, the system uses the **Char2** field as an index to an accent-table entry. When an accent key is pressed with an ALT GR key, the system uses the **Char3** field as an index to an accent-table entry. The system then waits for the next key. If the next key does not specify accent keys in the corresponding **AccentFlags** field or the next key is not found in the **Map[20][2]** field of the accent-table entry, then the character specified by the **NonAccent** field is used for the accent key and the second key is translated normally. Both characters are passed to the keyboard-input buffer after the "not-an-accent" beep sounds.

If a key does not change when a left or right SHIFT key is held down, it should use the same value for **Char1** and **Char2** so that the accent will apply in both the shifted and non-shifted cases. If the accent value is undefined when used with a SHIFT key or with the ALT GR key, the value in **Char2** or **Char3** should be zero.

If an accent key does not have ALT or CONTROL key mapping, the **AltAccent** and **CtlAccent** fields should be set to zero.

5.2.3.12 Shift Key (Type 0x000C)

A shift key (type 0x000C) represents a shift whose state changes when the key is pressed or released. The SHIFT and CONTROL keys are typical shift keys.

The **Char1** field holds a bit mask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. When the user presses the key, the system sets the shift-status word, and then clears it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a shift key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

5.2.3.13 General Toggle Key (Type 0x000D)

A general toggle key (type 0x000D) represents a shift key whose state changes when the key is pressed but not when it is released. The CAPSLOCK key is a typical toggle key.

The **Char1** field holds a bit mask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. The system uses **Char1** to set the lower byte of the shift-status word when the user first presses the key. Thereafter the system alternates between setting and clearing on each subsequent press. The system uses **Char2** to set the upper-byte word when the user presses the key and to clear it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a toggle key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

5.2.3.14 ALT Key (Type 0x000E)

The ALT key (type 0x000E) represents a special shift key that works in combination with the keypad keys to build character values. The ALT key requires its own key type so that the system knows to clear the accumulated value when the user begins to build a character using the keypad. Otherwise, the system treats the ALT key the same as any other shift key.

The **Char1** field holds a bit mask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. When the user presses the key, the system sets the shift-status word and clears it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a shift key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

If the **XTableFlags1** field specifies an ALT GR key, the ALT key may be treated as that key.

5.2.3.15 NUMLOCK Key (Type 0x000F)

The NUMLOCK key (type 0x000F) represents a special toggle key that, when pressed in combination with the CONTROL key, directs the system to temporarily stop screen output. Otherwise, the system treats the NUMLOCK the same as any other toggle key. When CONTROL+NUMLOCK stops screen output, the next keystroke (if it generates a valid character) restores output.

The **Char1** field holds a bit mask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. The system uses **Char1** to set the lower byte of the shift-status word when the user first presses the key. Thereafter the system alternates between setting and clearing on each press. The system uses **Char2** to set the upper-byte word when the user presses the key and to clear it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a toggle key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

5.2.3.16 CAPSLOCK Key (Type 0x0010)

The CAPSLOCK key (type 0x0010) represents a special toggle key. This key type only applies when the **XTableFlags1** field specifies that the CAPSLOCK key is to be processed like a SHIFTLOCK key. When processed as a SHIFTLOCK key, the CAPSLOCK key sets the keyboard shift-status word but cannot be used to clear the word. To do this, a SHIFT key must be pressed.

The **Char1** field holds a bit mask that the system uses to set the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. The system uses **Char1** to set the lower byte of the shift-status word when the user first presses the key. Thereafter the system clears the byte only if the user presses a SHIFT key. The system uses **Char2** to set the upper-byte word when the user presses the key and to clear it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a toggle key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

5.2.3.17 SCROLL LOCK Key (Type 0x0011)

The SCROLL LOCK key (type 0x0011) represents a special toggle key that generates a CONTROL+BREAK signal for a program when it is pressed with the CONTROL key. Otherwise, the system treats the SCROLL LOCK key the same as any other toggle key.

The **Char1** field holds a bit mask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bit mask that the system uses to set or clear the upper byte of the system's shift-status word. The system uses **Char1** to set the lower byte of the shift-status word when the user first presses the key. Thereafter the system alternates between setting and clearing on each press. The system uses **Char2** to set the upper-byte word when the user presses the key and to clear it when the user releases the key. If a secondary-key prefix (0xE0) is received immediately prior to a toggle key, the **Char3** field is used in place of **Char2** to set or clear the shift-status word.

5.2.3.18 Extended-Shift Key (Type 0x0012)

An extended-shift key (type 0x0012) represents a shift key that is used in conjunction with national-language support. The key is similar to the shift key (type 0x000C) but sets or clears the extra national-language-support byte of the keyboard-status word.

The character fields are defined as follows:

Field	Description
Char1	Specifies the bit mask in which the bits that are on define the field used for the **Char2** value. Only the bits in the national-language-support shift-status byte that correspond to the bits in this byte will be altered by the **Char2** value.

Field	Description
Char2	Specifies the bit mask used to set or clear bits in the extended-status byte when the key is pressed or released.
Char3	Specifies the replacement bit mask for **Char2** when the secondary key prefix (0xE0) is recognized immediately prior to this key being pressed.

Char1 and **Char2** can define single shift-status bits to set, clear, or toggle. **Char2** can be a set of coded bits (delineated by **Char1**) that will be set to a numeric value when the key is pressed and cleared to zero when released. When **Char1** has all bits on, the whole byte can be set to **Char2**.

5.2.3.19 Extended-Toggle Key (Type 0x0013)

An extended-toggle key (type 0x0013) represents a shift key that is used in conjunction with national-language support. The key is similar to the toggle key (type 0x000D) but it sets or clears the extra national-language-support byte of the keyboard-status word.

The character fields are defined as follows:

Field	Description
Char1	Specifies the bit mask in which the bits that are on define the field used for the **Char2** value. Only the bits in the national-language-support shift-status byte that correspond to the bits in this byte will be altered by the **Char2** value.
Char2	Specifies the bit mask used to set or clear bits in the extended-status byte when the key is pressed.
Char3	Specifies the replacement bit mask for **Char2** when the secondary-key prefix (0xE0) is recognized immediately prior to this key being pressed.

Char1 and **Char2** can define single shift-status bits to set, clear, or toggle. **Char2** can be a set of coded bits (delineated by **Char1**) that will be set to a numeric value when the key is pressed and set to zero when released. When **Char1** has all bits on, the whole byte can be set to **Char2**.

5.2.3.20 Special Foreign Key (Type 0x0014)

A special foreign key (type 0x0014) represents any character that may need a CAPSLOCK translation.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	**Char4**

Shift key	Field used
CAPSLOCK and SHIFT	**Char5**
CONTROL	Computed ASCII control value.
ALT	No effect.
ALT-GR	**Char3**

5.2.3.21 Special Foreign Key (Type 0x0015)

A special foreign key (type 0x0015) represents any character that may need an ALT translation.

Shift key	Field used
None	**Char1**
SHIFT	**Char2**
CAPSLOCK	No effect.
CONTROL	Computed ASCII control value.
ALT	**Char4**
ALT-GR	**Char3**

When ALT or ALT+SHIFT is pressed, the scan code and translated character code are equal.

5.3 Video Modes and Fonts

This section provides brief descriptions of the device-dependent values that may be used with the MS OS/2 video functions. In particular, it describes screen modes, screen attributes, video fonts, and physical-screen buffer addresses for the following display adapters:

IBM Monochrome/Printer Adapter
IBM Color Graphics Adapter (CGA)
IBM Enhanced Graphics Adapter (EGA)
IBM PS/2 Video Graphics Array (VGA)
IBM PS/2 Display Adapter

5.3.1 Screen Modes

The **VioSetMode** function sets the screen mode for the display adapter. The screen mode defines the type of output (text or graphics) and the resolution of the output; that is, it defines the width and height of the screen in character cells or pels. The available screen modes depend on the display's device driver as well as on the display adapter. Not all screen modes for a given display adapter are supported by the corresponding MS OS/2 display device driver. In general, an MS OS/2 display device driver supports at least one text mode and one graphics mode and, in many cases, the device driver supports all modes.

Tables 5.4 and 5.5 list the screen modes available for the IBM Monochrome/Printer Adapter, Color Graphics Adapter, Enhanced Graphics Adapter, Video Graphics Array, PS/2 Display Adapter, and any adapter that is one-hundred percent compatible with these.

Table 5.4 Text Modes

Columns	Rows	Colors	Cell width	Cell height	Vertical resolution	Horizontal resolution	Display
80	25	2	9	14	720	350	Monochrome/Printer Adapter
80	25	2	9	16	720	400	VGA, PS/2 Display Adapter
40	25	16	8	8	320	200	CGA,* EGA, VGA, PS/2 Display Adapter
40	25	16	8	14	320	350	EGA, VGA, and PS/2 Display Adapter
40	25	16	9	16	360	400	VGA, PS/2 Display Adapter
80	25	16	8	8	640	200	CGA,* EGA, VGA, PS/2 Display Adapter
80	25	16	8	14	640	350	EGA, VGA, PS/2 Display Adapter
80	25	16	9	16	720	400	VGA, PS/2 Display Adapter

Note * The color burst is turned off on the CGA.

Table 5.5 Graphics Modes

Colors	Vertical resolution	Horizontal resolution	Display
4	320	200	CGA,* EGA, VGA, and PS/2 Display Adapter
2	640	200	CGA,* EGA, VGA, and PS/2 Display Adapter
16	320	200	EGA, VGA, PS/2 Display Adapter
16	640	200	EGA, VGA, PS/2 Display Adapter
2	640	350	EGA, VGA, PS/2 Display Adapter
16	640	350	EGA,** VGA, PS/2 Display Adapter
2	640	480	VGA, PS/2 Display Adapter
16	640	480	VGA, PS/2 Display Adapter
256	320	200	VGA, PS/2 Display Adapter

Note * The color burst is turned off on the CGA.

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

When the screen is in graphics mode, MS OS/2 supports only the following **Vio** functions:

VioRegister
VioDeRegister
VioGetPhysBuf
VioSavRedrawWait
VioSavRedrawUndo
VioScrLock
VioScrUnLock
VioPopUp
VioEndPopUp
VioModeWait
VioModeUndo
VioGetFont (request type 1 only)
VioGetConfig
VioSetState (request types 0 and 1 only)
VioGetState (request types 0 and 1 only)
VioSetMode
VioGetMode

5.3.2 Screen Attributes

The screen attributes define the background and foreground colors and appearance of text when the screen is in text mode. A screen attribute is an 8-bit bit mask whose fields define the color and intensity of a character, as well as other attributes, such as underlining and blinking. The **VioWrtCellStr**, **VioWrtCharStrAtt**, **VioWrtNAttr**, and **VioWrtNCell** functions use screen attributes as input parameters. The meaning of the fields within a screen-attribute bit mask depends on the display adapter.

For the Monochrome/Printer Adapter, the screen attribute can be a combination of the following values:

Value	Meaning
0x00	Blank character
0x01	Underlined character
0x07	Normal character
0x08	High-intensity character
0x70	Reverse-video character
0x80	Blinking character or high-intensity background (depends on whether display-adapter blinker is active)

For the Color Graphics Adapter and the Enhanced Graphics Adapter, the screen attribute can be a combination of the following values:

Value	Meaning
0x00	Black character
0x01	Blue character
0x02	Green character
0x04	Red character
0x08	High-intensity character
0x10	Blue background
0x20	Green background
0x40	Red background
0x80	Blinking character

5.3.3 Physical-Screen Buffer Addresses

The physical-screen buffer address is the starting address of the display adapter's video-buffer memory. This starting address, as well as the size of the video memory and the format and meaning of the contents of the memory, depends on the display adapter and the screen mode.

5.3.4 Video Fonts

The **VioGetFont** and **VioSetFont** functions retrieve and set video fonts for the text-mode screen. These functions can be used with displays, such as the Enhanced Graphics Adapter and the Video Graphics Array, that accept downloadable fonts. To use a custom font, a program can either create it or modify a copy of an existing font. A program uses the **VioSetFont** function to set the current font and the **VioGetFont** function to copy existing fonts from the display.

For the Enhanced Graphics Adapter and Video Graphics Array, a video font is an array of 256 character cells. Each cell consists of an array of scan-line data. The cell height specifies number of scan lines for each cell. The width of the cell specifies the number of bytes for each scan line. Each bit represents a single pel in the character cell. If the bit is 1, the pel is the foreground color. If the bit is 0, the pel is the background color.

Some VGA text modes specify character widths of 9 pels. The video fonts used with this mode supply only 8 bits. The display provides the additional background pel automatically.

5.4 Resource-File Formats

An application can access the resources of an application or dynamic-link library by using the **DosGetResource** function. MS OS/2 has several predefined resource formats that Presentation Manager applications can use to create pointers, icons, bitmaps, menus, accelerator tables, and dialog windows. Other MS OS/2 programs can also access these resources directly, or they can define and access their own resources. The following is a list of the predefined resource formats:

Resource type	Resource format
RT_POINTER	Mouse-pointer shape
RT_BITMAP	Bitmap
RT_MENU	Menu template
RT_DIALOG	Dialog template
RT_STRING	String tables
RT_FONTDIR	Font directory
RT_FONT	Font
RT_ACCELTABLE	Accelerator tables
RT_RCDATA	Binary data
RT_MESSAGE	Error-message tables
RT_DLGINCLUDE	Dialog-include filename
RT_VKEYTBL	Scan-code to virtual-key tables
RT_KEYTBL	Key to font-glyph tables
RT_CHARTBL	Glyph to character tables
RT_DISPLAYINFO	Screen-display information

Predefined resources such as pointers, dialog windows, and fonts can be created using Presentation Manager applications such as Icon Editor, Dialog Editor, and Font Editor. Other resources can be generated by using the MS OS/2 Resource Compiler (**rc**). Resource Compiler also adds resources to the executable file for applications and dynamic-link libraries.

Presentation Manager applications use the following functions to retrieve resources from an application's executable file or a dynamic-link library. Some functions carry out additional steps, such as creating windows and bitmaps, and do not provide direct access to the data loaded.

- **GpiLoadBitmap**
- **GpiLoadFonts**
- **WinLoadPointer**
- **WinLoadMenu**
- **WinLoadDlg**
- **WinLoadAccelTable**
- **WinLoadMessage**
- **WinLoadString**

The following sections describe the internal format of the predefined resources. The format descriptions are useful for MS OS/2 programs that create new resources or that load these resources directly by using the **DosGetResource** function.

5.4.1 Pointer and Icon Resources

The RT_POINTER resource represents a pointer or icon resource. A pointer or icon resource is a special bitmap that contains two bit masks. Presentation Manager applications use the resource to draw mouse pointers or icons on the display. The **WinLoadPointer** function is typically used to load a pointer or icon resource and create a pointer handle. An application can draw the pointer or icon by passing the pointer handle to the **WinDrawPointer** function.

The pointer and icon resources have the following format:

```
/* These fields are identical to the BITMAPFILEHEADER structure. */

USHORT usType;              /* PT for pointer or IC for icon       */
ULONG cbSize;               /* size of resource (in bytes)         */
USHORT xHotspot;            /* x-coordinate of hot spot            */
USHORT yHotspot;            /* y-coordinate of hot spot            */
ULONG offBits;              /* offset to abANDMask array           */

/* These fields are identical to the BITMAPINFOHEADER structure. */

ULONG cbFix;                /* size of BITMAPINFOHEADER structure */
USHORT cx;                  /* width of bitmap (in pels)           */
USHORT cy;                  /* height of bitmap (in pels)          */
USHORT cPlanes;             /* count of color planes in bitmaps    */
USHORT cBitCount;           /* count of bits per pel               */

/* These fields define the masks and mask colors. */

RGB argbColor[1];           /* array of RGB colors                 */
BYTE abANDMask[1];          /* array for AND mask                  */
BYTE abXORMask[1];          /* array for XOR mask                  */
```

The only difference between resources is the **usType** field. For icon resources this field is set to IC; for pointer resources the field is PT.

The size of the **argbColor, abANDMask**, and **abXORMask** fields depends on the number of color planes and bits per pel specified by the **cPlanes** and **cBitCount** fields. The size of each bit mask also depends on the width and height of the bitmap. The bytes of the **abXORMask** field start immediately after the last byte in **abANDMask.**

Icon Editor can be used to create pointers and icons. The **POINTER** and **ICON** statements in Resource Compiler use the pointer and icon files created by Icon Editor to generate pointer and icon resources.

5.4.2 Bitmap Format

The RT_BITMAP resource represents a bitmap. Presentation Manager applications typically load the bitmap by using the **GpiLoadBitmap** function. This function returns a handle to the bitmap. An application can use the **GpiSetBitmap** function subsequently to set the bitmap as the current bitmap of a memory device context.

A bitmap resource has the following format:

```
/* These fields are identical to the BITMAPFILEHEADER structure. */

USHORT usType;               /* BM                              */
ULONG cbSize;                /* size of resource (in bytes)     */
USHORT xHotspot;             /* x-coordinate of hot spot        */
USHORT yHotspot;             /* y-coordinate of hot spot        */
ULONG offBits;               /* offset to abBitmap array        */

/* These fields are identical to the BITMAPINFOHEADER structure. */

ULONG cbFix;                 /* size of BITMAPINFOHEADER structure */
USHORT cx;                   /* width of bitmap (in pels)       */
USHORT cy;                   /* height of bitmap (in pels)      */
USHORT cPlanes;              /* count of color planes in bitmaps */
USHORT cBitCount;            /* count of bits per pel           */

/* These fields define the bitmap and its colors. */

RGB argbColor[1];            /* array of RGB colors             */
BYTE abBitmap[1];            /* array for bitmap bits           */
```

The size of the **argbColor** and **abBitmap** fields depends on the number of color planes and bits per pel specified by the **cPlanes** and **cBitCount** fields. The size of the **abBitmap** field also depends on the width and height of the bitmap.

Icon Editor can be used to create bitmaps. The **BITMAP** statement in Resource Compiler uses the bitmap files created by Icon Editor to generate bitmap resources.

5.4.3 String and Message Resources

The RT_STRING or RT_MESSAGE resource is a table of exactly 16 character strings representing error messages and other text used by an application. Presentation Manager applications typically load individual strings from a table by using the **WinLoadString** or **WinLoadMessage** function. These functions use a string identifier to determine the table containing the string and the string's location in the table.

Each string or message resource consists of a table of exactly 16 entries. Each entry has the following form:

```
BYTE cchText;        /* length of string including zero terminator */
SZ szText[cchText];  /* zero-terminated string                     */
```

String and message tables have resource identifiers starting at 1. Each string also has a unique identifier. A string's identifier determines which table the string is in and where in the table it is located. The following C-language expressions specify the location of a string:

```
USHORT idString;       /* string ID                            */
USHORT idTable;        /* resource ID of string or message table */
USHORT iString;        /* index in table of string             */

idTable = (idString / 16) + 1;
iString = idString % 16;
```

For example, if the string identifier is 1, the string is in table 1 at entry 1. If the string identifier is 17, the string is in table 2 at entry 1.

The **STRINGTABLE** and **MESSAGETABLE** statements in Resource Compiler generate string and message resources.

5.4.4 Menu Resource

The RT_MENU resource represents a menu template. A menu template contains all the data needed to create a menu. A Presentation Manager application typically loads a menu-template resource by using the **WinLoadMenu** function.

A menu-template resource has the following format:

```
ULONG cbSize;                /* size of menu template (in bytes)  */
USHORT idCodePage;           /* code page for menu names          */
USHORT idClass;              /* menu window-class ID              */
USHORT cItems;               /* number of items in menu           */

/* These fields are repeated for each item. */

USHORT fStyle;               /* menu-style flags                  */
USHORT fAttributes;          /* menu-attribute flags              */
USHORT cmd;                  /* menu-item ID                      */
SZ szItemName[1];            /* null-terminated menu name         */
```

If a menu item is a submenu, its fields are followed immediately by the menu-template resource that defines the menu items in that submenu.

The length of the **szItemName** field is variable and depends on the menu item. If the menu item has no name, for example, if it is a menu separator, no **szItemName** field is given.

The **MENU** statement in Resource Compiler generates menu templates.

5.4.5 Accelerator-Table Resource

The RT_ACCELTABLE resource represents a keyboard-accelerator table. Accelerator tables are used by Presentation Manager applications to translate keystrokes into commands; that is, they translate WM_CHAR messages into WM_COMMAND, WM_SYSCOMMAND, or WM_HELP messages. An application typically loads accelerator tables by using the **WinLoadAccelTable** function.

The accelerator-table resource has the following format:

```
/* These fields are identical to the ACCELTABLE structure. */

USHORT cAccel;            /* number of accelerators in the table */
USHORT codepage;          /* code page for text                  */

/* These fields are identical to the ACCEL structure. */

USHORT fs;                /* accelerator flags                   */
USHORT key;               /* keystroke to be translated          */
USHORT cmd;               /* command ID of translated keystroke  */
```

The fields defining the keystroke and command are repeated for each accelerator in the table. The **fs** field specifies whether the **key** field represents a virtual key, a scan code, or a key combination.

The **ACCELTABLE** statement in Resource Compiler generates accelerator-table resources.

5.0.1 Dialog Templates

The RT_DIALOG resource represents a dialog-template resource. A dialog-template resource contains all the data needed to create a dialog window and corresponding child controls. Presentation Manager applications typically use the **WinLoadDlg** or **WinDlgBox** function to load the resource. The function creates the dialog window and control windows specified by the template.

Some applications load the resource directly by using the **DosGetResource** function. Loading a dialog-template resource directly allows an application to examine and modify the data before creating the dialog window. The application can then pass the data to the **WinCreateDlg** function to create the dialog window, or extract individual parameters from the data and pass the parameters to functions such as **WinCreateWindow** to create other types of windows.

A dialog-template resource has the following form:

```
/* These fields are identical to the DLGTEMPLATE structure. */

USHORT      cbTemplate;       /* number of bytes in the template   */
USHORT      type;             /* dialog type                       */
USHORT      codepage;         /* code-page for text                */
USHORT      offadlgti;        /* offset to 1st dialog item (12)    */
USHORT      fsTemplateStatus; /* template-status flags             */
USHORT      iItemFocus;       /* index to initial focus window     */
USHORT      coffPresParams;   /* offset to presentation parameters */
DLGTITEM    adlgti[1]         /* array of DLGTITEM structures      */

/* These fields are identical to the DLGTITEM structure. */

USHORT    fsItemStatus;     /* item-status flags                   */
USHORT    cChildren;        /* number of child windows             */
USHORT    cchClassName;     /* number of characters in class name  */
USHORT    offClassName;     /* offset to class name or class ID    */
USHORT    cchText;          /* number of characters in window text */
USHORT    offText;          /* offset to window text               */
ULONG     flStyle;          /* window styles                       */
SHORT     x;                /* x-coordinate of window              */
SHORT     y;                /* y-coordinate of window              */
SHORT     cx;               /* width of window                     */
SHORT     cy;               /* height of window                    */
USHORT    id;               /* window ID                           */
USHORT    offPresParams;    /* offset to presentation parameters   */
USHORT    offCtlData;       /* offset to class-specific data       */
```

The fields defining the dialog items are repeated for each window in the template. Data such as class name and window text appears after the fields for the last window. If a window has child windows, the fields of the child windows immediately follow the fields for the parent window. If the **cchClassName** field is zero, the **offClassName** field must contain a valid window-class identifier. The format of the class-specific data depends on the window class. In general, the first word of the presentation parameter data and the class-specific data must specify the length of that data in bytes.

Dialog Box Editor can be used to create dialog-template resources. The Resource Compiler statements **DLGTEMPLATE** and **WINDOWTEMPLATE** generate dialog-template resources.

5.0.2 Dialog-Include Resource

The RT_DLGINCLUDE resource is a filename. This resource typically is used in conjunction with a dialog-template resource that has the same resource identifier. The dialog-include resource specifies the include file that contains definitions for constants used in the dialog template. Although the resource is useful to Dialog Box Editor, other applications probably will not need it.

The **DLGINCLUDE** statement in Resource Compiler generates dialog-include resources.

5.0.3 Font Resource

The RT_FONT resource represents a font resource. A font resource consists of the font metrics and character data that describe a font. Presentation Manager applications load font resources by using the **GpiLoadFont** function. This function makes all font resources in a specified dynamic-link library available to the application.

A font resource is identical in format to a font file. For more information, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

Font Editor can be used to create fonts. The **FONT** statement in Resource Compiler uses the font created by Font Editor to generate font resources.

5.0.4 Font-Directory Resource

The RT_FONTDIR resource represents a font directory. A font directory consists of the font metrics of a corresponding font resource. MS OS/2 uses font directories to load information about a font without having to load the entire font into memory.

The font-directory resource has the following form:

```
USHORT usFontDir;    /* resource type (always 6)                 */
USHORT cFonts;       /* count of fonts in directory              */
USHORT cbSize;       /* size of each directory entry (in bytes) */

/* These fields are repeated for each font. */

USHORT idFont;        /* resource ID for corresponding font     */
FOCAMETRICS foca;     /* font metrics from font file            */
```

The **FONTDIR** statement in Resource Compiler generates a font-directory resource. The FONT statement of Resource Compiler also generates a font directory. It does this as it generates the font resource, so the FONTDIR statement is rarely used.

5.0.5 Binary Data

The RT_RCDATA resource represent one or more bytes of binary data. The binary can have any format. The application defines the content of the data.

The **RCDATA** statement in Resource Compiler generates binary-data resources.

5.4.11 MS OS/2 Internal Resources

The RT_VKEYTBL, RT_KEYTBL, RT_CHARTBL, and RT_DISPLAYINFO resources represent data used internally by MS OS/2. MS OS/2 uses this data to carry out system-level tasks—for example, translating scan codes to virtual keys and translating code points in a code page to font gylphs.

Appendixes

Appendix

A

Error Values

A.1 Introduction

This chapter contains the possible error values that can be returned by the MS OS/2 base system functions. Before you can use these errors in your application, you must define the INCL_BASE, INCL_ERRORS, or INCL_DOSERRORS constant before including the *os2.h* file. The following code is a typical example:

```
#define INCL_DOS
#define INCL_DOSERRORS

#include <os2.h>
```

A.2 Errors

The following list gives the error values that may be returned by the **Dos**, **Kbd**, **Mou**, and **Vio** functions. The error values are listed in numerical order, and the corresponding error constant is given for each value.

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
7	ERROR_ARENA_TRASHED
8	ERROR_NOT_ENOUGH_MEMORY
9	ERROR_INVALID_BLOCK
10	ERROR_BAD_ENVIRONMENT
11	ERROR_BAD_FORMAT
12	ERROR_INVALID_ACCESS
13	ERROR_INVALID_DATA
15	ERROR_INVALID_DRIVE
16	ERROR_CURRENT_DIRECTORY
17	ERROR_NOT_SAME_DEVICE
18	ERROR_NO_MORE_FILES
19	ERROR_WRITE_PROTECT
20	ERROR_BAD_UNIT
21	ERROR_NOT_READY
22	ERROR_BAD_COMMAND
23	ERROR_CRC
24	ERROR_BAD_LENGTH
25	ERROR_SEEK
26	ERROR_NOT_DOS_DISK
27	ERROR_SECTOR_NOT_FOUND
28	ERROR_OUT_OF_PAPER
29	ERROR_WRITE_FAULT
30	ERROR_READ_FAULT
31	ERROR_GEN_FAILURE
32	ERROR_SHARING_VIOLATION
33	ERROR_LOCK_VIOLATION
34	ERROR_WRONG_DISK
35	ERROR_FCB_UNAVAILABLE
36	ERROR_SHARING_BUFFER_EXCEEDED
50	ERROR_NOT_SUPPORTED
80	ERROR_FILE_EXISTS
81	ERROR_DUP_FCB
82	ERROR_CANNOT_MAKE
83	ERROR_FAIL_I24
84	ERROR_OUT_OF_STRUCTURES
85	ERROR_ALREADY_ASSIGNED
86	ERROR_INVALID_PASSWORD
87	ERROR_INVALID_PARAMETER
88	ERROR_NET_WRITE_FAULT
89	ERROR_NO_PROC_SLOTS
90	ERROR_NOT_FROZEN
91	ERR_TSTOVFL
92	ERR_TSTDUP
93	ERROR_NO_ITEMS
95	ERROR_INTERRUPT
100	ERROR_TOO_MANY_SEMAPHORES
101	ERROR_EXCL_SEM_ALREADY_OWNED
102	ERROR_SEM_IS_SET
103	ERROR_TOO_MANY_SEM_REQUESTS
104	ERROR_INVALID_AT_INTERRUPT_TIME
105	ERROR_SEM_OWNER_DIED
106	ERROR_SEM_USER_LIMIT
107	ERROR_DISK_CHANGE
108	ERROR_DRIVE_LOCKED
109	ERROR_BROKEN_PIPE
110	ERROR_OPEN_FAILED
111	ERROR_BUFFER_OVERFLOW
112	ERROR_DISK_FULL
113	ERROR_NO_MORE_SEARCH_HANDLES
114	ERROR_INVALID_TARGET_HANDLE
115	ERROR_PROTECTION_VIOLATION
116	ERROR_VIOKBD_REQUEST
117	ERROR_INVALID_CATEGORY
118	ERROR_INVALID_VERIFY_SWITCH
119	ERROR_BAD_DRIVER_LEVEL
120	ERROR_CALL_NOT_IMPLEMENTED
121	ERROR_SEM_TIMEOUT
122	ERROR_INSUFFICIENT_BUFFER
123	ERROR_INVALID_NAME
124	ERROR_INVALID_LEVEL
125	ERROR_NO_VOLUME_LABEL
126	ERROR_MOD_NOT_FOUND
127	ERROR_PROC_NOT_FOUND
128	ERROR_WAIT_NO_CHILDREN
129	ERROR_CHILD_NOT_COMPLETE
130	ERROR_DIRECT_ACCESS_HANDLE
131	ERROR_NEGATIVE_SEEK
132	ERROR_SEEK_ON_DEVICE
133	ERROR_IS_JOIN_TARGET
134	ERROR_IS_JOINED
135	ERROR_IS_SUBSTED
136	ERROR_NOT_JOINED
137	ERROR_NOT_SUBSTED
138	ERROR_JOIN_TO_JOIN
139	ERROR_SUBST_TO_SUBST
140	ERROR_JOIN_TO_SUBST
141	ERROR_SUBST_TO_JOIN
142	ERROR_BUSY_DRIVE
143	ERROR_SAME_DRIVE
144	ERROR_DIR_NOT_ROOT
145	ERROR_DIR_NOT_EMPTY
146	ERROR_IS_SUBST_PATH
147	ERROR_IS_JOIN_PATH
148	ERROR_PATH_BUSY
149	ERROR_IS_SUBST_TARGET
150	ERROR_SYSTEM_TRACE
151	ERROR_INVALID_EVENT_COUNT
152	ERROR_TOO_MANY_MUXWAITERS
153	ERROR_INVALID_LIST_FORMAT
154	ERROR_LABEL_TOO_LONG
155	ERROR_TOO_MANY_TCBS
156	ERROR_SIGNAL_REFUSED
157	ERROR_DISCARDED
158	ERROR_NOT_LOCKED
159	ERROR_BAD_THREADID_ADDR
160	ERROR_BAD_ARGUMENTS
161	ERROR_BAD_PATHNAME
162	ERROR_SIGNAL_PENDING
163	ERROR_UNCERTAIN_MEDIA
164	ERROR_MAX_THRDS_REACHED
165	ERROR_MONITORS_NOT_SUPPORTED
166	ERROR_UNC_DRIVER_NOT_INSTALLED
167	ERROR_LOCK_FAILED
168	ERROR_SWAPIO_FAILED

169	ERROR_SWAPIN_FAILED
170	ERROR_BUSY
180	ERROR_INVALID_SEGMENT_NUMBER
181	ERROR_INVALID_CALLGATE
182	ERROR_INVALID_ORDINAL
183	ERROR_ALREADY_EXISTS
184	ERROR_NO_CHILD_PROCESS
185	ERROR_CHILD_ALIVE_NOWAIT
186	ERROR_INVALID_FLAG_NUMBER
187	ERROR_SEM_NOT_FOUND
188	ERROR_INVALID_STARTING_CODESEG
189	ERROR_INVALID_STACKSEG
190	ERROR_INVALID_MODULETYPE
191	ERROR_INVALID_EXE_SIGNATURE
192	ERROR_EXE_MARKED_INVALID
193	ERROR_BAD_EXE_FORMAT
194	ERROR_ITERATED_DATA_EXCEEDS_64K
195	ERROR_INVALID_MINALLOCSIZE
196	ERROR_DYNLINK_FROM_INVALID_RING
197	ERROR_IOPL_NOT_ENABLED
198	ERROR_INVALID_SEGDPL
199	ERROR_AUTODATASEG_EXCEEDS_64k
200	ERROR_RING2SEG_MUST_BE_MOVABLE
201	ERROR_RELOC_CHAIN_XEEDS_SEGLIM
202	ERROR_INFLOOP_IN_RELOC_CHAIN
203	ERROR_ENVVAR_NOT_FOUND
204	ERROR_NOT_CURRENT_CTRY
205	ERROR_NO_SIGNAL_SENT
206	ERROR_FILENAME_EXCED_RANGE
207	ERROR_RING2_STACK_IN_USE
208	ERROR_META_EXPANSION_TOO_LONG
209	ERROR_INVALID_SIGNAL_NUMBER
210	ERROR_THREAD_1_INACTIVE
211	ERROR_INFO_NOT_AVAIL
212	ERROR_LOCKED
213	ERROR_BAD_DYNALINK
214	ERROR_TOO_MANY_MODULES
215	ERROR_NESTING_NOT_ALLOWED
230	ERROR_BAD_PIPE
231	ERROR_PIPE_BUSY
232	ERROR_NO_DATA
233	ERROR_PIPE_NOT_CONNECTED
234	ERROR_MORE_DATA
240	ERROR_VC_DISCONNECTED
303	ERROR_INVALID_PROCID
304	ERROR_INVALID_PDELTA
305	ERROR_NOT_DESCENDANT
306	ERROR_NOT_SESSION_MANAGER
307	ERROR_INVALID_PCLASS
308	ERROR_INVALID_SCOPE
309	ERROR_INVALID_THREADID
310	ERROR_DOSSUB_SHRINK
311	ERROR_DOSSUB_NOMEM
312	ERROR_DOSSUB_OVERLAP
313	ERROR_DOSSUB_BADSIZE
314	ERROR_DOSSUB_BADFLAG
315	ERROR_DOSSUB_BADSELECTOR
316	ERROR_MR_MSG_TOO_LONG
317	ERROR_MR_MID_NOT_FOUND
318	ERROR_MR_UN_ACC_MSGF
319	ERROR_MR_INV_MSGF_FORMAT
320	ERROR_MR_INV_IVCOUNT
321	ERROR_MR_UN_PERFORM
322	ERROR_TS_WAKEUP
323	ERROR_TS_SEMHANDLE
324	ERROR_TS_NOTIMER
326	ERROR_TS_HANDLE
327	ERROR_TS_DATETIME
328	ERROR_SYS_INTERNAL
329	ERROR_QUE_CURRENT_NAME
330	ERROR_QUE_PROC_NOT_OWNED
331	ERROR_QUE_PROC_OWNED
332	ERROR_QUE_DUPLICATE
333	ERROR_QUE_ELEMENT_NOT_EXIST
334	ERROR_QUE_NO_MEMORY
335	ERROR_QUE_INVALID_NAME
336	ERROR_QUE_INVALID_PRIORITY
337	ERROR_QUE_INVALID_HANDLE
338	ERROR_QUE_LINK_NOT_FOUND
339	ERROR_QUE_MEMORY_ERROR
340	ERROR_QUE_PREV_AT_END
341	ERROR_QUE_PROC_NO_ACCESS
342	ERROR_QUE_EMPTY
343	ERROR_QUE_NAME_NOT_EXIST
344	ERROR_QUE_NOT_INITIALIZED
345	ERROR_QUE_UNABLE_TO_ACCESS
346	ERROR_QUE_UNABLE_TO_ADD
347	ERROR_QUE_UNABLE_TO_INIT
349	ERROR_VIO_INVALID_MASK
350	ERROR_VIO_PTR
351	ERROR_VIO_APTR
352	ERROR_VIO_RPTR
353	ERROR_VIO_CPTR
354	ERROR_VIO_LPTR
355	ERROR_VIO_MODE
356	ERROR_VIO_WIDTH
357	ERROR_VIO_ATTR
358	ERROR_VIO_ROW
359	ERROR_VIO_COL
360	ERROR_VIO_TOPROW
361	ERROR_VIO_BOTROW
362	ERROR_VIO_RIGHTCOL
363	ERROR_VIO_LEFTCOL
364	ERROR_SCS_CALL
365	ERROR_SCS_VALUE
366	ERROR_VIO_WAIT_FLAG
367	ERROR_VIO_UNLOCK
368	ERROR_SGS_NOT_SESSION_MGR
369	ERROR_SMG_INVALID_SGID
369	ERROR_SMG_INVALID_SESSION_ID
370	ERROR_SMG_NOSG
370	ERROR_SMG_NO_SESSIONS
371	ERROR_SMG_GRP_NOT_FOUND
371	ERROR_SMG_SESSION_NOT_FOUND
372	ERROR_SMG_SET_TITLE
373	ERROR_KBD_PARAMETER
374	ERROR_KBD_NO_DEVICE
375	ERROR_KBD_INVALID_IOWAIT
376	ERROR_KBD_INVALID_LENGTH
377	ERROR_KBD_INVALID_ECHO_MASK

378	ERROR_KBD_INVALID_INPUT_MASK
379	ERROR_MON_INVALID_PARMS
380	ERROR_MON_INVALID_DEVNAME
381	ERROR_MON_INVALID_HANDLE
382	ERROR_MON_BUFFER_TOO_SMALL
383	ERROR_MON_BUFFER_EMPTY
384	ERROR_MON_DATA_TOO_LARGE
385	ERROR_MOUSE_NO_DEVICE
386	ERROR_MOUSE_INV_HANDLE
387	ERROR_MOUSE_INV_PARMS
388	ERROR_MOUSE_CANT_RESET
389	ERROR_MOUSE_DISPLAY_PARMS
390	ERROR_MOUSE_INV_MODULE
391	ERROR_MOUSE_INV_ENTRY_PT
392	ERROR_MOUSE_INV_MASK
393	NO_ERROR_MOUSE_NO_DATA
394	NO_ERROR_MOUSE_PTR_DRAWN
395	ERROR_INVALID_FREQUENCY
396	ERROR_NLS_NO_COUNTRY_FILE
397	ERROR_NLS_OPEN_FAILED
398	ERROR_NLS_NO_CTRY_CODE
398	ERROR_NO_COUNTRY_OR_CODEPAGE
399	ERROR_NLS_TABLE_TRUNCATED
400	ERROR_NLS_BAD_TYPE
401	ERROR_NLS_TYPE_NOT_FOUND
402	ERROR_VIO_SMG_ONLY
403	ERROR_VIO_INVALID_ASCIIZ
404	ERROR_VIO_DEREGISTER
405	ERROR_VIO_NO_POPUP
406	ERROR_VIO_EXISTING_POPUP
407	ERROR_KBD_SMG_ONLY
408	ERROR_KBD_INVALID_ASCIIZ
409	ERROR_KBD_INVALID_MASK
410	ERROR_KBD_REGISTER
411	ERROR_KBD_DEREGISTER
412	ERROR_MOUSE_SMG_ONLY
413	ERROR_MOUSE_INVALID_ASCIIZ
414	ERROR_MOUSE_INVALID_MASK
415	ERROR_MOUSE_REGISTER
416	ERROR_MOUSE_DEREGISTER
417	ERROR_SMG_BAD_ACTION
418	ERROR_SMG_INVALID_CALL
419	ERROR_SCS_SG_NOTFOUND
420	ERROR_SCS_NOT_SHELL
421	ERROR_VIO_INVALID_PARMS
422	ERROR_VIO_FUNCTION_OWNED
423	ERROR_VIO_RETURN
424	ERROR_SCS_INVALID_FUNCTION
425	ERROR_SCS_NOT_SESSION_MGR
426	ERROR_VIO_REGISTER
427	ERROR_VIO_NO_MODE_THREAD
428	ERROR_VIO_NO_SAVE_RESTORE_THD
429	ERROR_VIO_IN_BG
430	ERROR_VIO_ILLEGAL_DURING_POPUP
431	ERROR_SMG_NOT_BASESHELL
432	ERROR_SMG_BAD_STATUSREQ
433	ERROR_QUE_INVALID_WAIT
434	ERROR_VIO_LOCK
435	ERROR_MOUSE_INVALID_IOWAIT
436	ERROR_VIO_INVALID_HANDLE
437	ERROR_VIO_ILLEGAL_DURING_LOCK
438	ERROR_VIO_INVALID_LENGTH
439	ERROR_KBD_INVALID_HANDLE
440	ERROR_KBD_NO_MORE_HANDLE
441	ERROR_KBD_CANNOT_CREATE_KCB
442	ERROR_KBD_CODEPAGE_LOAD_INCOMPL
443	ERROR_KBD_INVALID_CODEPAGE_ID
444	ERROR_KBD_NO_CODEPAGE_SUPPORT
445	ERROR_KBD_FOCUS_REQUIRED
446	ERROR_KBD_FOCUS_ALREADY_ACTIVE
447	ERROR_KBD_KEYBOARD_BUSY
448	ERROR_KBD_INVALID_CODEPAGE
449	ERROR_KBD_UNABLE_TO_FOCUS
450	ERROR_SMG_SESSION_NON_SELECT
451	ERROR_SMG_SESSION_NOT_FOREGRND
452	ERROR_SMG_SESSION_NOT_PARENT
453	ERROR_SMG_INVALID_START_MODE
454	ERROR_SMG_INVALID_RELATED_OPT
455	ERROR_SMG_INVALID_BOND_OPTION
456	ERROR_SMG_INVALID_SELECT_OPT
457	ERROR_SMG_START_IN_BACKGROUND
458	ERROR_SMG_INVALID_STOP_OPTION
459	ERROR_SMG_BAD_RESERVE
460	ERROR_SMG_PROCESS_NOT_PARENT
461	ERROR_SMG_INVALID_DATA_LENGTH
462	ERROR_SMG_NOT_BOUND
463	ERROR_SMG_RETRY_SUB_ALLOC
464	ERROR_KBD_DETACHED
465	ERROR_VIO_DETACHED
466	ERROR_MOU_DETACHED
467	ERROR_VIO_FONT
468	ERROR_VIO_USER_FONT
469	ERROR_VIO_BAD_CP
470	ERROR_VIO_NO_CP
471	ERROR_VIO_NA_CP
472	ERROR_INVALID_CODE_PAGE
473	ERROR_CPLIST_TOO_SMALL
474	ERROR_CP_NOT_MOVED
475	ERROR_MODE_SWITCH_INIT
476	ERROR_CODE_PAGE_NOT_FOUND
477	ERROR_UNEXPECTED_SLOT_RETURNED
478	ERROR_SMG_INVALID_TRACE_OPTION
479	ERROR_VIO_INTERNAL_RESOURCE
480	ERROR_VIO_SHELL_INIT
481	ERROR_SMG_NO_HARD_ERRORS
482	ERROR_CP_SWITCH_INCOMPLETE
483	ERROR_VIO_TRANSPARENT_POPUP
484	ERROR_CRITSEC_OVERFLOW
485	ERROR_CRITSEC_UNDERFLOW
486	ERROR_VIO_BAD_RESERVE
487	ERROR_INVALID_ADDRESS
488	ERROR_ZERO_SELECTORS_REQUESTED
489	ERROR_NOT_ENOUGH_SELECTORS_AVA
490	ERROR_INVALID_SELECTOR
491	ERROR_SMG_INVALID_PROGRAM_TYPE
492	ERROR_SMG_INVALID_PGM_CONTROL
493	ERROR_SMG_INVALID_INHERIT_OPT
494	ERROR_VIO_EXTENDED_SG
495	ERROR_VIO_NOT_PRES_MGR_SG
496	ERROR_VIO_SHIELD_OWNED
497	ERROR_VIO_NO_MORE_HANDLES
498	ERROR_VIO_SEE_ERROR_LOG
499	ERROR_VIO_ASSOCIATED_DC
500	ERROR_KBD_NO_CONSOLE

501	ERROR_MOUSE_NO_CONSOLE
502	ERROR_MOUSE_INVALID_HANDLE
503	ERROR_SMG_INVALID_DEBUG_PARMS
504	ERROR_KBD_EXTENDED_SG
505	ERROR_MOU_EXTENDED_SG
506	ERROR_SMG_INVALID_ICON_FILE
0xF000	ERROR_USER_DEFINED_BASE
0	ERROR_I24_WRITE_PROTECT
1	ERROR_I24_BAD_UNIT
2	ERROR_I24_NOT_READY
3	ERROR_I24_BAD_COMMAND
4	ERROR_I24_CRC
5	ERROR_I24_BAD_LENGTH
6	ERROR_I24_SEEK
7	ERROR_I24_NOT_DOS_DISK
8	ERROR_I24_SECTOR_NOT_FOUND
9	ERROR_I24_OUT_OF_PAPER
10	ERROR_I24_WRITE_FAULT
11	ERROR_I24_READ_FAULT
12	ERROR_I24_GEN_FAILURE
13	ERROR_I24_DISK_CHANGE
15	ERROR_I24_WRONG_DISK
16	ERROR_I24_UNCERTAIN_MEDIA
17	ERROR_I24_CHAR_CALL_INTERRUPTED
18	ERROR_I24_NO_MONITOR_SUPPORT
19	ERROR_I24_INVALID_PARAMETER

/* Values for error CLASS */

1	ERRCLASS_OUTRES
2	ERRCLASS_TEMPSIT
3	ERRCLASS_AUTH
4	ERRCLASS_INTRN
5	ERRCLASS_HRDFAIL
6	ERRCLASS_SYSFAIL
7	ERRCLASS_APPERR
8	ERRCLASS_NOTFND
9	ERRCLASS_BADFMT
10	ERRCLASS_LOCKED
11	ERRCLASS_MEDIA
12	ERRCLASS_ALREADY
13	ERRCLASS_UNK
14	ERRCLASS_CANT
15	ERRCLASS_TIME

/* Values for error ACTION */

1	ERRACT_RETRY
2	ERRACT_DLYRET
3	ERRACT_USER
4	ERRACT_ABORT
5	ERRACT_PANIC
6	ERRACT_IGNORE
7	ERRACT_INTRET

/* Values for error LOCUS */

1	ERRLOC_UNK
2	ERRLOC_DISK
3	ERRLOC_NET
4	ERRLOC_SERDEV
5	ERRLOC_MEM

/* */
/* intercomponent error codes */
/* (from 8000H or 32768) */
/* */

32768	ERROR_SWAPPER_NOT_ACTIVE
32769	ERROR_INVALID_SWAPID
32770	ERROR_IOERR_SWAP_FILE
32771	ERROR_SWAP_TABLE_FULL
32772	ERROR_SWAP_FILE_FULL
32773	ERROR_CANT_INIT_SWAPPER
32774	ERROR_SWAPPER_ALREADY_INIT
32775	ERROR_PMM_INSUFFICIENT_MEMORY
32776	ERROR_PMM_INVALID_FLAGS
32777	ERROR_PMM_INVALID_ADDRESS
32778	ERROR_PMM_LOCK_FAILED
32779	ERROR_PMM_UNLOCK_FAILED
32780	ERROR_PMM_MOVE_INCOMPLETE
32781	ERROR_UCOM_DRIVE_RENAMED
32782	ERROR_UCOM_FILENAME_TRUNCATED
32783	ERROR_UCOM_BUFFER_LENGTH
32784	ERROR_MON_CHAIN_HANDLE
32785	ERROR_MON_NOT_REGISTERED
32786	ERROR_SMG_ALREADY_TOP
32787	ERROR_PMM_ARENA_MODIFIED
32788	ERROR_SMG_PRINTER_OPEN
32789	ERROR_PMM_SET_FLAGS_FAILED
32790	ERROR_INVALID_DOS_DD
65026	ERROR_CPSIO_CODE_PAGE_INVALID
65027	ERROR_CPSIO_NO_SPOOLER
65028	ERROR_CPSIO_FONT_ID_INVALID
65033	ERROR_CPSIO_INTERNAL_ERROR
65034	ERROR_CPSIO_INVALID_PTR_NAME
65037	ERROR_CPSIO_NOT_ACTIVE
65039	ERROR_CPSIO_PID_FULL
65040	ERROR_CPSIO_PID_NOT_FOUND
65043	ERROR_CPSIO_READ_CTL_SEQ
65045	ERROR_CPSIO_READ_FNT_DEF
65047	ERROR_CPSIO_WRITE_ERROR
65048	ERROR_CPSIO_WRITE_FULL_ERROR
65049	ERROR_CPSIO_WRITE_HANDLE_BAD
65074	ERROR_CPSIO_SWIT_LOAD
65077	ERROR_CPSIO_INV_COMMAND
65078	ERROR_CPSIO_NO_FONT_SWIT

Appendix

B

ANSI Escape Sequences

B.1 Introduction

This appendix lists all the escape sequences that can be used in the functions such as **DosWrite** and **VioWrtTTY** to control the operation of the screen while in text mode. The escape sequences can be used in family API, advanced video-input-and-output (AVIO) and full-screen programs.

The ANSI escape sequences affect cursor positioning, erase functions, and screen graphics. The sequences must be typed exactly as shown with all parameters replaced with appropriate values. No spaces are allowed. The ESC in the syntax represents the escape character (27).

B.2 Cursor Functions

The following functions affect the movement of the cursor.

B.2.1 Cursor Position

ESC[*row***;***col***H**

or

ESC[*row***;***col***f**

These two sequences move the cursor to the position specified by the parameters. When no parameters are provided, the cursor moves to the home position (the upper-left corner of the screen).

B.2.2 Cursor Up

ESC[*n***A**

This sequence moves the cursor up *n* rows without changing columns. If the cursor is already on the top line, MS OS/2 ignores this sequence.

B.2.3 Cursor Down

ESC[*n***B**

This sequence moves the cursor down *n* rows without changing columns. If the cursor is already on the bottom row, MS OS/2 ignores this sequence.

B.2.4 Cursor Forward

ESC[*n***C**

This sequence moves the cursor forward *n* columns without changing lines. If the cursor is already in the far-right column, MS OS/2 ignores this sequence.

B.2.5 Cursor Backward

ESC[*n*D

This sequence moves the cursor back *n* columns without changing lines. If the cursor is already in the far-left column, MS OS/2 ignores this sequence.

B.2.6 Save Cursor Position

ESC[s

This sequence saves the current cursor position. This position can be restored with the Restore Cursor Position sequence.

B.2.7 Restore Cursor Position

ESC[u

This sequence restores the cursor position to the Save Cursor Position value.

B.3 Erase Functions

The following functions erase the screen.

B.3.1 Erase Display

ESC[2J

This sequence erases the screen and moves the cursor to the home position (the upper-left corner of the screen).

B.3.2 Erase Line

ESC[K

This sequence erases from the cursor to the end of the line (including the cursor position).

B.4 Screen Graphics Functions

The following functions affect screen graphics.

B.4.1 Set Graphics Rendition

ESC[*g*; ... ;*g*m

This sequence calls the graphics functions specified by the following numeric values. These functions remain until the next occurrence of this sequence. This sequence works only if the screen device supports graphics.

The *g* variable may be any of the following values:

Value	Function
0	All attributes off
1	Bold on
2	Faint on
3	Italic on
5	Blink on
6	Rapid-blink on
7	Reverse video on
8	Concealed on
30	Black foreground
31	Red foreground
32	Green foreground
33	Yellow foreground
34	Blue foreground
35	Magenta foreground
36	Cyan foreground
37	White foreground
40	Black background
41	Red background
42	Green background
43	Yellow background
44	Blue background
45	Magenta background
46	Cyan background
47	White background
48	Subscript
49	Superscript

The values 30 through 47 meet the ISO 6429 standard.

B.4.2 Set Mode

ESC[=*s*h

This sequence changes the screen width or type. The *s* variable can be one of the following numeric values:

Value	Function
0	40 × 25 black and white
1	40 × 25 color
2	80 × 25 black and white
3	80 × 25 color
4	320 × 200 color
5	320 × 200 black and white
6	640 × 200 black and white
7	Wraps at the end of each line

B.4.3 Reset Mode

ESC[=*s*l

The values for this escape sequence are the same as for Set Mode, except that the value 7 resets the mode that causes wrapping at the end of each line.

Appendix

C

Country and Code-Page Information

C.1 Introduction

MS OS/2 supports multiple countries and languages, allowing for customization. This appendix lists the countries and languages supported by MS OS/2 and gives the related country and keyboard codes. The five supported code pages are also given.

C.2 Supported Countries

MS OS/2 supports these countries:

Country	Country code	Keyboard code
United States	001	US
Canada (French)	002	CF
Latin America	003	LA
Netherlands	031	NL
Belgium	032	BE
France	033	FR
Spain	034	SP
Italy	039	IT
Switzerland (French)	041	SF
Switzerland (German)	041	SG
United Kingdom	044	UK
Denmark	045	DK
Sweden	046	SV
Norway	047	NO
Germany	049	GR
Australia	061	—
Portugal	351	PO
Finland	358	SU

C.3 Code Pages

A code page is a set of symbols used to display text. Each symbol represents a letter, digit, punctuation mark, or other character found in written languages. Each symbol in a code page is identified by a unique value called a code point. A program displays a given symbol by supplying its corresponding code point.

MS OS/2 provides predefined code pages. Each code page, identified by a unique number, contains a set of symbols for a given written language. For example, code page 860 contains the symbols needed to display messages in Portuguese.

MS OS/2 supports the following five code pages:

437 United States

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p	Ç	É	á	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q	ü	æ	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r	é	Æ	ó	▓	┬	╥	Γ	≥
-3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	Θ	∙
-A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
-B	♂	←	+	;	K	[	k	{	ï	¢	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|	î	£	¼	╝	╠	▄	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}	ì	¥	¡	╜	═	▌	φ	²
-E	♫	▲	.	>	N	^	n	~	Ä	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂	Å	ƒ	»	┐	╧	▀	∩	

850 Multilingual

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p	Ç	É	á	░	└	ð	Ó	-
-1	☺	◄	!	1	A	Q	a	q	ü	æ	í	▒	┴	Đ	ß	±
-2	☻	↕	"	2	B	R	b	r	é	Æ	ó	▓	┬	Ê	Ô	‗
-3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	Ë	Ò	¾
-4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	È	õ	¶
-5	♣	§	%	5	E	U	e	u	à	ò	Ñ	Á	┼	ı	Õ	§
-6	♠	▬	&	6	F	V	f	v	å	û	ª	Â	ã	Í	µ	÷
-7	•	↨	'	7	G	W	g	w	ç	ù	º	À	Ã	Î	þ	¸
-8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	©	╚	Ï	Þ	°
-9	○	↓	)	9	I	Y	i	y	ë	Ö	®	╣	╔	┘	Ú	¨
-A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Û	·
-B	♂	←	+	;	K	[	k	{	ï	ø	½	╗	╦	█	Ù	¹
-C	♀	∟	,	<	L	\	l	|	î	£	¼	╝	╠	▄	ý	³
-D	♪	↔	-	=	M	]	m	}	ì	Ø	¡	¢	═	¦	Ý	²
-E	♫	▲	.	>	N	^	n	~	Ä	×	«	¥	╬	Ì	¯	■
-F	☼	▼	/	?	O	_	o	⌂	Å	ƒ	»	┐	¤	▀	´	

860 Portuguese

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p	Ç	É	á	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q	ü	À	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r	é	È	ó	▓	┬	╥	Γ	≥
-3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t	ã	õ	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v	Á	Ú	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x	ê	Ì	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y	Ê	Õ	Ò	╣	╔	┘	Θ	•
-A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
-B	♂	←	+	;	K	[	k	{	Í	¢	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|	Ô	£	¼	╝	╠	▄	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}	ì	Ù	¡	╜	═	▌	φ	²
-E	♫	▲	.	>	N	^	n	~	Ã	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂	Â	Ó	»	┐	╧	▀	∩	

863 French-Canadian

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p	Ç	É	¦	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q	ü	È	´	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r	é	Ê	ó	▓	┬	╥	Γ	≥
-3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t	Â	Ë	¨	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u	à	Ï	¸	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v	¶	û	³	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w	ç	ù	¯	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x	ê	¤	Î	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y	ë	Ô	⌐	╣	╔	┘	Θ	∙
-A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
-B	♂	←	+	;	K	[	k	{	ï	¢	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|	î	£	¼	╝	╠	▄	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}	‗	Ù	¾	╜	═	▌	φ	²
-E	♫	▲	.	>	N	^	n	~	À	Û	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂	§	ƒ	»	┐	╧	▀	∩	

865 Nordic

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-	8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p	Ç	É	á	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q	ü	æ	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r	é	Æ	ó	▓	┬	╥	Γ	≥
-3	♥	‼	#	3	C	S	c	s	â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t	ä	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u	à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v	å	û	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w	ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x	ê	ÿ	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y	ë	Ö	⌐	╣	╔	┘	Θ	∙
-A	◙	→	*	:	J	Z	j	z	è	Ü	¬	║	╩	┌	Ω	·
-B	♂	←	+	;	K	[	k	{	ï	ø	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|	î	£	¼	╝	╠	▄	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}	ì	Ø	¡	╜	═	▌	φ	²
-E	♫	▲	.	>	N	^	n	~	Ä	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂	Å	ƒ	¤	┐	╧	▀	∩	

Index

A

B

C

D

E

F

G

H

I

K

L

M

Also Available From Microsoft Press

Authoritative Information for OS/2 Programmers

INSIDE OS/2

Gordon Letwin

"The best way to understand the overall philosophy of OS/2 will be to read this book."

— ***Bill Gates***

Here — from Microsoft's Chief Architect of Systems Software — is an exciting technical examination of the philosophy, key development issues, programming implications, and role of OS/2 in the office of the future. And Letwin provides the first in-depth look at each of OS/2's design elements. This is a valuable and revealing programmer-to-programmer discussion of the graphical user interface, multitasking, memory management, protection, encapsulation, interprocess communication, and direct device access. You can't get a more inside view.

304 pages, 7 3/8 x 9 1/4, softcover, \$19.95.
[Order Code 86-96288]

ADVANCED OS/2 PROGRAMMING

Ray Duncan

Authoritative information, expert advice, and great assembly-language code make this comprehensive overview of the features and structure of OS/2 indispensable to any serious OS/2 programmer. Duncan addresses a range of significant OS/2 issues: programming the user interface; mass storage; memory management; multitasking; interprocess communications; customizing filters, device drivers, and monitors; and using OS/2 dynamic link libraries. A valuable reference section includes detailed information on each of the more than 250 system service calls in version 1.1 of the OS/2 kernel.

800 pages, 7 3/8 x 9 1/4, softcover, \$24.95
[Book Code 86-96106]

PROGRAMMING THE OS/2 PRESENTATION MANAGER

Charles Petzold

New! Here is the first full discussion of the features and operation of the OS/2 1.1 Presentation Manager. If you're developing OS/2 applications, this book will guide you through Presentation Manager's system of windows, messages, and function calls. Petzold includes scores of valuable C programs and utilities.

Endorsed by the Microsoft Systems Software group, this book is unparalleled for its clarity, detail, and comprehensiveness. Petzold covers: managing windows ■ handling input and output ■ controlling child windows ■ using bitmaps, icons, pointers, and strings ■ accessing the menu and keyboard accelerators ■ working with dialog boxes ■ understanding dynamic linking ■ and more.

864 pages, 7⅜ x 9¼, softcover, $29.95
[Order Code 86-96791]

ESSENTIAL OS/2 FUNCTIONS: Programmer's Quick Reference

Ray Duncan

Concise information on the essential OS/2 function calls within the application program interface (API). Entries are included for all kernel API functions for OS/2 version 1.0: Dos, Kbd, Mou, and Vio. Brief descriptions of each function are included, as well as a list of the required parameters, returned results, programming notes and warnings, family API call identification, and error codes. Conveniently arranged to provide quick access to the information you need.

172 pages, 4¾ x 8, softcover, $9.95
[Order Code 86-96866]

For the Windows Programmer

PROGRAMMING WINDOWS

Charles Petzold

Your fastest route to successful application programming with Windows. Full of indispensable reference data, tested programming advice, and page after page of creative sample programs and utilities. Topics include getting the most out of the keyboard, mouse, and timer; working with icons, cursors, bitmaps, and strings; exploiting Windows' memory management; creating menus; taking advantage of child window controls; incorporating keyboard accelerators; using dynamically linkable libraries; and mastering the Graphics Device Interface (GDI). A thorough, up-to-date, and authoritative look at Windows' rich graphical environment.

864 pages, 7⅜ x 9¼
$24.95 (sc) [Order Code 86-96049]
$34.95 (hc) [Order Code 86-96130]

Programmer's Quick Reference Series

MS-DOS® FUNCTIONS

Ray Duncan

The kind of information every seasoned programmer needs right at hand. Includes detailed information on MS-DOS system service calls, along with valuable programming notes. Covers MS-DOS through version 4.

128 pages, 4¾ x 8, softcover, $5.95
[Order Code 86-96411]

IBM® ROM BIOS

Ray Duncan

Essential for every assembly-language or C programmer at any experience level. Designed for quick and easy access to information, this guide includes all the core information on each of the ROM BIOS services.

128 pages, 4¾ x 8, softcover, $5.95
[Order Code 86-96478]

MS-DOS® EXTENSIONS

Ray Duncan

Brings together the hard-to-find programming information on the Lotus®/Intel®/Microsoft® Expanded Memory Specification (EMS) version 4.0, the Lotus/Intel/Microsoft/AST Extended Memory Specification (XMS) version 2.0, the Microsoft CD-ROM Extensions version 2.1, and the Microsoft Mouse driver, version 6. An overview of each function is accompanied by a list of its required parameters, returned results, and applicable programming notes.

128 pages, 4¾ x 8, softcover, $6.95
[Order Code 86-97229]

Solid Language References

MICROSOFT® C: SECRETS, SHORTCUTS & SOLUTIONS

Kris Jamsa

Here is a fact-filled, example-packed resource for any current or aspiring Microsoft C programmer working in the DOS environment. Each chapter highlights specific C programming facts, tips, and traps so that key information or items of special interest are immediately accessible. Hundreds of short sample programs support Jamsa's instruction and encourage experimentation.

If you're new to C, Microsoft C, or even Microsoft QuickC, Jamsa's fast-paced, highly readable style will help you quickly master the fundamentals. If you're a seasoned programmer, you'll find page after page of advanced information that will hone your programming skills and make your Microsoft C programs fast, clean, and efficient. Jamsa shows you how to:

access the DOS command line ■ expand wildcard characters into matching filenames ■ use I/O redirection ■ master dynamic memory allocation ■ take advantage of C's predefined global variables ■ optimize your programs for increased speed ■ enhance your program's video appearance ■ make full use of the MAKE and LIB tools

500 pages, 7³/₈ x 9¹/₄, softcover, $24.95
[Order Code 86-97112]

PROFICIENT C

Augie Hansen

''*A beautifully-conceived text, clearly written and logically organized... a superb guide.*'' ***Computer Book Review***

An information-packed handbook for intermediate to advanced DOS programmers that includes dozens of file-oriented and screen-oriented C programs and specially developed utilities. A successful blend of programming advice and practical example programs.

512 pages, 7³/₈ x 9¹/₄, softcover, $22.95
[Order Code 86-95710]

VARIATIONS IN C

Steve Schustack

Foreword by Gerald Weinberg

A superb guide for experienced programmers who want to develop efficient, portable, high-quality application software using C in the DOS environment. In addition to an overview of the basic syntax of C, Schustack provides valuable techniques for structured programming. A complete, 1500-line source code sample program illustrates key topics. Special comments and cautions are highlighted throughout.

368 pages, 7³/₈ x 9¹/₄, softcover, $19.95
[Order Code 86-95249]

STANDARD C: Programmer's Quick Reference

P.J. Plauger and Jim Brodie

All the basic information needed to read and write Standard C programs that conform to the recently approved ANSI and ISO standard for the C programming language. Scores of diagrams illustrate the syntax rules. Whether you're new to C or familiar with an earlier dialect, this will prove a handy companion.

224 pages, 4¾ x 8, softcover, $7.95
[Order Code 86-96676]

MICROSOFT® QUICKC PROGRAMMING

The Waite Group

Your springboard to the core of the Microsoft QuickC. This book is loaded with practical information and advice on *every* element of QuickC, along with hundreds of specially constructed listings. Included are the tools to help you master QuickC's built-in libraries; manage file input and output; work with strings, arrays, pointers, structures, and unions; use the graphics modes; develop and link large C programs; and debug your source code.

624 pages, 7⅜ x 9¼, softcover, $19.95
[Order Code 86-96114]

MICROSOFT® QUICKBASIC®, 2nd ed.

Douglas Hergert

"No matter what your level of programming experience, you'll find this book irreplaceable when you start to program in QuickBASIC." **Online Today**

Here's a great introduction to all the development tools, features, and user-interface enhancements in Microsoft QuickBASIC. And there's more — six specially designed, full-length programs including a database manager, an information-gathering and data-analysis program, and a chart program that reenforce solid structured programming techniques.

464 pages, 7⅜ x 9¼, softcover, $19.95
[Order Code 86-96387]

80387 math coprocessor, the protection scheme, the implementation of a virtual memory system through paging, and compatibility with earlier Intel microprocessors. Of special note is the comprehensive, clearly organized instruction set reference — guaranteed to be a valuable resource.

464 pages, 7³/₈ x 9¹/₄, softcover, $24.95
[Order Code 86-96494]

THE PROGRAMMER'S PC SOURCEBOOK

Thom Hogan

At last! A reference to save you the time required to find key pieces of technical data. Here is important factual information — previously published in scores of other sources — organized into one convenient reference. Focusing on IBM PCs and compatibles, PS/2s and MS-DOS, the hundreds of charts and tables cover: ■ numeric conversions and character sets ■ DOS commands and utilities ■ DOS function calls and support tables ■ DOS BIOS calls and support tables ■ other interrupts, mouse, and EMS support ■ Microsoft Windows ■ keyboards, video adapters, and peripherals ■ chips, jumpers, switches, and registers ■ hardware descriptions ■ and more.

560 pages, 8¹/₂ x 11, softcover, $24.95
[Order Code 86-96296]

THE *NEW* PETER NORTON PROGRAMMER'S GUIDE TO THE IBM® PC & PS/2®

Peter Norton and Richard Wilton

A must-have classic on mastering the inner workings of IBM micros — now completely updated to include the PS/2 line. Sharpen your programming skills and learn to create simple, clean, portable programs with this successful combination of astute programming advice, proven techniques, and solid technical data. Covers 8088, 80286 and 80386 microprocessors; ROM BIOS basics and ROM BIOS services; video, disk and keyboard basics; DOS basics, interrupts, and functions (through version 4); interrupts, device drivers, and video programming. Accept no substitutes; this is the book to have.

528 pages, 7³/₈ x 9¹/₄, softcover, $22.95
[Order Code 86-96635]

The Microsoft Press CD-ROM Library

THE MICROSOFT® CD-ROM YEARBOOK: 1989/1990

Microsoft Press

Foreword by Bill Gates

A dynamic, fact-filled portrait and analysis of the wide-ranging, fast-paced CD-ROM industry. Indispensable for anyone involved in the industry as well as an information-packed compendium for those curious about CD-ROM. Readers can use the book as a valuable sourcebook of facts, statistics, and forecasts, or dip into it for fascinating articles, reviews, and analyses of the industry. Articles include:

- an absorbing history — in text and pictures — of the CD-ROM industry
- reviews of products — hardware and software — considered outstanding or standard-setting
- profiles of the leading companies and people in the industry
- an overview of the process of developing a CD-ROM product
- a review of the legal issues of protection, rights and permissions, contracts and royalties surrounding CD-ROM publishing
- the strategies and pitfalls involved in getting a CD-ROM product to market

The breadth of accurate, up-to-date information in THE MICROSOFT CD-ROM YEARBOOK is impressive including:

- comprehensive reference listings of the people, equipment, available titles, sources, and resources in the CD ROM industry
- a glossary of industry terms
- a calendar of industry events and conferences
- specialized bibliographies

This is *the* reference of fact and opinion on the industry.

960 pages, 7³/₈ x 9¹/₄, softcover, $79.95
[Order Code 86-97203]

CD ROM: THE NEW PAPYRUS

Edited by Steve Lambert and Suzanne Ropiequet

"This 619-page compendium, with contributions from more than 30 optical-memory specialists, promises to become the bible of CD ROM." **David Bunnell, Macworld**

This special compendium of 45 articles by leading authorities examines every facet of compact disc read only memory technology: hardware, software, applications, publishing systems, marketing, and the user interface. Includes introductory as well as technical information.

608 pages, 7³/₈ x 9¹/₄, softcover, $21.95
[Order Code 86-95454]

CD ROM 2: OPTICAL PUBLISHING

Edited by Suzanne Ropiequet with John Einberger and Bill Zoellick

"Recommended reading for any information professional." **Online Today**

This is a comprehensive overview of the entire optical publishing process. Topics include: evaluating and defining storage and retrieval methods; collecting, preparing, and indexing data; updating strategies; data protection and copyrighting; and more. Plus information on the High Sierra Logical Format. In addition, the editors trace the development of two CD ROM projects from initial concept to final product. For publishers, technical managers, and entrepreneurs.

384 pages, 7³/₈ x 9¹/₄, softcover, $22.95
[Order Code 86-95686]

INTERACTIVE MULTIMEDIA

Foreword by John Sculley

Edited by Sueanne Ambron and Kristina Hooper

Apple Computer Corp. brought together leading researchers and developers to produce this informative collection of 21 articles. The result is a sourcebook of ideas and inspiration for software and hardware developers, educators, publishers, and information providers. The contributors, including Doug Englebart, Sam Gibbon, and Peter Cook, represent the industries — computers, television, and publishing — whose products will provide the content and media for education in the future. Filled with examples and pilot projects that define the new meaning of multimedia. Published with Apple Computer, Inc.

352 pages, 7³/₈ x 9¹/₄, softcover, $24.95
[Order Code 86-96379]

MICROSOFT®

OS/2 Programmer's Reference

The Microsoft® Operating System/2 Programmer's Reference Library should be the cornerstone of every OS/2 developer's programming library. These volumes are required references for professional developers creating applications for the retail market; for corporate programmers creating in-house software programs; for hardware manufacturers creating software to support their products; and for all other experienced programmers working in the OS/2 environment.

Each volume in the series is written by a team of OS/2 specialists—many involved in the development and ongoing enhancement of OS/2 at Microsoft. These books provide in-depth, accurate, and up-to-date information from the Microsoft OS/2 Presentation Manager Toolkit—the software development kit essential for creating OS/2 applications.

Volume 1

Volume 1 details the conceptual framework of the MS® OS/2 Application Programming Interface (API). Included are thorough descriptions of MS® OS/2 programming models, overviews of basic programming considerations, and explanations of the interaction between the API and the rest of the MS® OS/2 system. Sections include *Introducing MS® OS/2, Window Manager, Graphics Programming Interface,* and *System Services.*

Volume 2

Volume 2 is a comprehensive, alphabetic listing of MS® OS/2 Presentation Manager functions as well as the structures and file formats used with these functions. Each function entry includes information on syntax; descriptions of the function's actions and purpose; parameters and field definitions; return values, error values, and restrictions; source-code examples; and programming notes. Appendix included.

Volume 3

Similar in format to Volume 2, Volume 3 is a comprehensive alphabetic listing of MS® OS/2 base functions, including their structures and file formats. Appendixes included.

U.S.A.	**$19.95**
U.K.	£18.95
Austral.	$29.95

(recommended)

ISBN 1-55615-222-1

51995

9 781556 152221

Computers/Operating Systems/OS/2