

Microsoft[®]

Operating System/2

Programmer's Toolkit

Programmer's Reference

Version 1.0

Microsoft Corporation

Information in this document is subject to change without notice and does not represent a commitment on the part of Microsoft Corporation. The software and/or databases described in this document are furnished under a license agreement or nondisclosure agreement. The software and/or databases may be used or copied only in accordance with the terms of the agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual and/or database may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without the written permission of Microsoft Corporation.

© Copyright Microsoft Corporation, 1988. All rights reserved.
Simultaneously published in the U.S. and Canada.

Microsoft®, MS®, MS-DOS®, and the Microsoft logo are registered trademarks of Microsoft Corporation.

Intel® is a registered trademark of Intel Corporation.

IBM® and PC/AT® are registered trademarks, and Personal System/2[™] is a trademark, of International Business Machines Corporation.

Document No. 060060014-100-R00-0388
Part No. 01887

Contents

1 Introduction 1

- 1.1 Overview 3
- 1.2 MS OS/2 Functions 4
- 1.3 Naming Conventions 6
- 1.4 Notational Conventions 9
- 1.5 Other References 10

2 Overview 11

- 2.1 Introduction 13
- 2.2 Multitasking Functions 14
- 2.3 Memory-Management Functions 16
- 2.4 Dynamic-Link Functions 19
- 2.5 File-System Functions 19
- 2.6 Input-and-Output Control Functions 24
- 2.7 Keyboard Functions 25
- 2.8 Mouse Functions 26
- 2.9 Video I/O Functions 28
- 2.10 Pipe Functions 30
- 2.11 Queue Functions 30
- 2.12 Semaphore Functions 31
- 2.13 Signal Functions 32
- 2.14 Timer Functions 33
- 2.15 Device-Monitor Functions 34
- 2.16 Error-Handling Functions 35
- 2.17 Program-Startup Functions 36
- 2.18 Miscellaneous Functions 38
- 2.19 Family Application Programming Interface 38

3 Functions Directory 45

- 3.1 Introduction 47
- 3.2 Functions 47

4 Input-and-Output Control Functions 391

- 4.1 Introduction 393
- 4.2 Category and Function Codes 393

- 4.3 Physical Disk Control 394
- 4.4 Category and Function Code 394
- 4.5 Functions 398

A Utility Macros 517

- A.1 Introduction 519
- A.2 Macros 519

B Devices 525

- B.1 Introduction 527
- B.2 Screen Modes 527
- B.3 Screen Attributes 528
- B.4 Physical Screen Buffer Addresses 529

C Code Pages 531

- C.1 Introduction 533
- C.2 Predefined Translation Tables 533
- C.3 Translation-Table Format 533
- C.4 Key Types 537

D Format of MS OS/2 Executable Files 555

- D.1 Introduction 557
- D.2 MS-DOS Executable Header 558
- D.3 New Executable Header 558
- D.4 Segment Table 561
- D.5 Resource Table 562
- D.6 Module-Reference Table 564
- D.7 Entry-Point Table 564
- D.8 Resident- and Nonresident-Name Tables 565
- D.9 Imported-Names Table 566
- D.10 Segments 566

E ANSI Escape Sequences 569

- E.1 Introduction 571
- E.2 Cursor Functions 571

E.3	Erase Functions	573
E.4	Screen Graphics Functions	573

Index	577
--------------	------------

Figures

Figure D.1	Executable-File Format	557
Figure D.2	Resource Table	562
Figure D.3	Resident- and Nonresident-Name Table Entry Format	565

Tables

Table 3.1	Open Modes for fOpenMode Parameter	160
Table 3.2	pbOutBuf	169
Table 3.3	pbParmBuf	169
Table 3.4	File-Handle State	181
Table 4.1	DTR or RTS: Input Handshaking	464
Table 4.2	RTS Only: Toggle on Transmit (Toggle)	465
Table B.1	Screen Modes for Monochrome/Printer Adapter	527
Table B.2	Screen Modes for CGA	528
Table B.3	Screen Modes for EGA	528
Table C.1	Shift Key Masks	539

Chapter 1

Introduction

1.1	Overview	3
1.2	MS OS/2 Functions	4
1.2.1	Bit Masks in Function Parameters	5
1.2.2	Structures	6
1.3	Naming Conventions	6
1.3.1	Parameter and Field Names	6
1.3.1.1	Prefixes	7
1.3.1.2	Base Types	8
1.3.2	Constant Names	9
1.4	Notational Conventions	9
1.5	Other References	10

10

11

12

1.1 Overview

This manual describes the Microsoft® Operating System/2 (MS® OS/2) system functions. MS OS/2 is a single-user, multitasking operating system for personal computers. MS OS/2 system functions let programs use the operating system to carry out tasks such as reading from and writing to disk files, allocating memory, and starting other programs.

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.

This chapter, “Introduction,” provides a brief description of MS OS/2 calling conventions and illustrates function calls in various languages.

Chapter 2, “Overview,” lists MS OS/2 function groups and explains the concepts, purpose, and use of the functions in each group.

Chapter 3, “Functions Directory,” is an alphabetical listing of MS OS/2 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 meaningful work.

Chapter 4, “Input-and-Output Control Functions,” lists the input-and-output control (IOctl) functions used to control input and output devices such as the serial ports, keyboard, and mouse.

Appendix A, “Utility Macros,” defines the utility macros provided with the MS OS/2 C-language header files.

Appendix B, “Devices,” supplies device-specific information, such as screen attributes, that can be used in some MS OS/2 functions.

Appendix C, “Code Pages,” defines code pages and provides the format for translation tables used to support code pages.

Appendix D, “Format of MS OS/2 Executable Files,” describes the MS OS/2 executable file format.

Appendix E, “ANSI Escape Sequences,” lists the escape sequences that can be used with MS OS/2.

This manual is intended to fully describe the MS OS/2 functions and the structures and file formats used with the functions. It does not show how to use these functions to carry out specific tasks, and it does not show how to write, compile, and link programs containing the functions. For more information on these topics, see the *Microsoft Operating System/2 Programmer’s Learning Guide* and *Microsoft Operating System/2 Programming Tools*.

1.2 MS OS/2 Functions

In MS OS/2, all programs request operating-system services by calling system functions. This is unlike MS-DOS® programs, which must request system services through a software interrupt (**int 21h**).

The MS OS/2 system functions let C, Pascal, and other high-level-language programs request services directly. MS OS/2 uses a high-level-language calling convention for all system functions, so a C, Pascal, or other high-level-language program can call an MS OS/2 system function just as if it were a C, Pascal, or other high-level-language function. In MS-DOS programs, these high-level-language programs must use an assembly-language routine to move parameters into registers and execute the **int 21h** instruction as required by MS-DOS. In MS OS/2, no assembly-language routine is needed.

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 **DosOpen** function in a C-language program. In C, the **DosOpen** function name and parameter types are defined in the *os2.h* 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         */
        0,             /* file's attribute         */
        0x10,          /* action to take if file exists */
        0x41,          /* file's open mode         */
        0L);           /* Reserved                 */
```

Note that the *os2.h* file uses the **pascal** keyword to define MS OS/2 functions. This ensures that the compiler generates the appropriate calling sequence for the MS OS/2 function.

To use MS OS/2 functions in other high-level languages or in assembly language, you need to know the MS OS/2 calling conventions.

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 right-to-left order. This means each function must have a fixed number of parameters. If a parameter specifies an address, the address must be a far address; that is, it must have the form *selector:offset*.
- The function automatically removes the parameters from the stack as it returns.
- You must use an inter-segment call instruction to call the function. This is a requirement for all dynamic-link-library functions.
- The function returns zero or an error code in the **ax** register, but preserves the value of all other registers except 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 to call the **DosOpen** function from an assembly-language program:

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

push      ds                      ; filename to open
push      ax, offset name
push      ds                      ; address of file handle
push      ax, offset hFile
push      ds                      ; address to store action taken
push      ax, offset usAction
push      0                      ; size of new file 0100H
push      100
push      0                      ; file's attribute
push      0010H                 ; action to take if file exists
push      0041H                 ; file's open mode
push      0                      ; Reserved
push      0
call      DOSOPEN
```

1.2.1 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 short or long value. Bit masks are a way to pack many Boolean flags (that is, 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 are typically 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 (either 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: 0x0001, 0x0002, and 0x0004. 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, the corresponding bit is set. Generally, the opposite result is assumed when the value is not used. For example, using 0x0002 in the **fbType** field enables graphics mode; not using it disables graphics mode.

1.2.2 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 header files provided with the Microsoft C compiler. Occasionally, an MS OS/2 function may have a structure which has no corresponding header-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 a header file.

1.3 Naming Conventions

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

1.3.1 Parameter and Field Names

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

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

1.3.1.1 Prefixes

The following is a list of standard prefixes:

Prefix	Description
<i>p</i>	A pointer. This prefix identifies a far, or 32-bit, pointer to a given item. For example, <i>pch</i> is a far pointer to a character.
<i>np</i>	A near pointer. This prefix identifies a near, or 16-bit, pointer to a given item. For example, <i>npch</i> is a near pointer to a character.
<i>a</i>	An array. This prefix identifies an array of two or more items of a given type. For example, <i>ach</i> is an array of characters.
<i>i</i>	An index into an array. For example, <i>ich</i> is an index to one character in an array of characters.
<i>c</i>	A 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, <i>cch</i> is a count of characters although it may be declared with the USHORT type.
<i>h</i>	A handle. This prefix is used for values that uniquely identify an object but that cannot be used to access the object directly. For example, <i>hfile</i> is a handle to a file.
<i>off</i>	An offset. This prefix is used for values that represent offsets from the beginning of a buffer or a structure.
<i>id</i>	An identifier. This prefix is used for values that identify an object. For example, <i>idSession</i> is a parameter that specifies a session identifier.

1.3.1.2 Base Types

The following is a list of standard base types:

Base Type	Type/Description
<i>f</i>	BOOL . A flag or Boolean variable. The qualifier should describe the condition associated with the flag when it is TRUE. For example, <i>fSuccess</i> is TRUE if successful, FALSE if not; <i>fError</i> is TRUE if an error occurs, and FALSE if no error occurs. For objects of type BOOL , value zero implies FALSE; any nonzero value implies TRUE.
<i>ch</i>	CHAR . Signed 8-bit quantity.
<i>s</i>	SHORT . Signed 16-bit quantity.
<i>l</i>	LONG . Signed 32-bit quantity.
<i>uch</i>	UCHAR . Unsigned 8-bit quantity.
<i>us</i>	USHORT . Unsigned 16-bit quantity.
<i>ul</i>	ULONG . Unsigned 32-bit quantity.
<i>b</i>	BYTE . Unsigned 8-bit quantity; a byte. Same as <i>uch</i> .
<i>sz</i>	CHAR[] . Array of characters, terminated with a null character (that is, the last byte is set to zero).
<i>fb</i>	UCHAR . Array of flags in a byte. Used when more than one flag is packed in an 8-bit value. Values for such an array are typically created by combining two or more values using the logical OR operator.
<i>fs</i>	USHORT . Array of flags in a short. Used when more than one flag is packed in a 16-bit value. Values for such an array are typically created by combining two or more values using the logical OR operator.
<i>fl</i>	ULONG . Array of flags in a long. Used when more than one flag is packed in a 32-bit value. Values for such an array are typically created by combining two or more values using the logical OR operator.
<i>sel</i>	SEL . 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 is a list of the base types for the structures described in this manual:

<i>ctryc</i>	<i>kbc</i>	<i>mourt</i>	<i>vioin</i>
<i>ctryi</i>	<i>kbsi</i>	<i>mouisc</i>	<i>vioint</i>
<i>date</i>	<i>kbst</i>	<i>mrs</i>	<i>viomi</i>
<i>fdate</i>	<i>kbzl</i>	<i>mzsl</i>	<i>vioos</i>
<i>findbuf</i>	<i>lis</i>	<i>resc</i>	<i>viopal</i>
<i>fsalloc</i>	<i>mouev</i>	<i>stdata</i>	<i>viopb</i>
<i>fsts</i>	<i>moupl</i>	<i>stdata</i>	
<i>ftime</i>	<i>moups</i>	<i>vioci</i>	
<i>gis</i>	<i>mouqi</i>	<i>viofi</i>	

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 using 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 an 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 is used for keywords, such as function, data-type, structure, and macro names. These names are spelled exactly as they should appear in source programs.
<i>italics</i>	Italics are used to indicate the name of an argument; this name must be replaced by an actual argument.
Monospace	Monospace type is used for example program code fragments.

1.5 Other References

For information on 80286 or 80386 architecture, please refer to the following books:

80286 and 80287 Programmer's Reference Manual, 1987, Intel Corporation.

80386 Programmer's Reference Manual, 1987, Intel Corporation.

80387 Programmer's Reference Manual, 1987, Intel Corporation.

The iAPX 286 Programmer's Reference Manual, 1987, Intel Corporation.

Chapter 2

Overview

2.1	Introduction	13
2.2	Multitasking Functions	14
2.3	Memory-Management Functions	16
2.4	Dynamic-Link Functions	19
2.5	File-System Functions	19
2.6	Input-and-Output Control Functions	24
2.7	Keyboard Functions	25
2.8	Mouse Functions	26
2.9	Video I/O Functions	28
2.10	Pipe Functions	30
2.11	Queue Functions	30
2.12	Semaphore Functions	31
2.13	Signal Functions	32
2.14	Timer Functions	33
2.15	Device-Monitor Functions	34
2.16	Error-Handling Functions	35
2.17	Program-Startup Functions	36
2.18	Miscellaneous Functions	38
2.19	Family Application Programming Interface	38

2.1 Introduction

MS OS/2 functions let programs use the operating system to carry out tasks such as controlling input and output, memory management, and program execution. MS OS/2 functions divide neatly by task into several groups. Each group of functions lets a program carry out a general task, such as using a keyboard, using disk files, or starting programs. The functions within a group provide specific support for carrying out or controlling the general task.

MS OS/2 has the following function groups:

- Multitasking
- Memory Management
- Dynamic Linking
- File System Input and Output
- Input-and-Output Control
- Keyboard Input and Output
- Mouse Input and Output
- Video Input and Output
- Pipes
- Queues
- Semaphores
- Signals
- Timers
- Device Monitors
- Error Handling
- Program Startup
- Miscellaneous
- Family Application Programming Interface

Each of the following sections in this chapter lists the functions in a group, describes the general task the group carries out, and explains how the individual function contributes to the task.

2.2 Multitasking Functions

MS OS/2 multitasking functions let a program start other programs or execute more than one copy of a program. Each program has access to all the computer's resources, such as memory, disk drives, screen, keyboard, and the CPU itself. The system carefully manages these resources so that programs can access them without conflict.

The following are multitasking functions:

DosCreateThread	DosGetPid
DosCWait	DosGetPrty
DosExecPgm	DosResumeThread
DosExit	DosSetPrty
DosExitList	DosSuspendThread

An MS OS/2 program that has been loaded into memory and prepared for execution is called a process. Each process has at least one thread, called the main thread or thread 1. The process consists of the program code, data, and other resources, such as files, pipes, and queues, that belong to the program. The thread consists of the current register values, the stack, and the state of execution of the program. When MS OS/2 executes a program, it makes sure that the process's code and data are in memory and the thread's registers and stack are set before it passes execution control.

A process can have more than one thread. Each thread runs independently, keeping its own register and execution state. Threads share the same data segment (that is, they share the program's globally-defined variables). Although a thread can execute any part of the program, including a part being executed by another thread, threads are typically used to execute separate parts. This distributes the available CPU time and lets a program carry out several tasks simultaneously, for example, loading a file and prompting the user for input at the same time. A process creates a thread by using the **DosCreateThread** function.

MS OS/2 has a system scheduler that determines how to distribute execution control among the programs currently running. The scheduler uses time slicing to distribute execution control. This means the scheduler periodically gives each thread in each process a small slice of CPU time. The thread executes until its time is up, and then the scheduler stops the thread and starts another.

The scheduler does not share CPU time equally among all threads. It uses a priority scheme to determine when a thread receives a time slice. The scheduler has three priority classes: time-critical, regular, and idle-time. A time-critical thread always receives a time slice before a regular thread, and a regular thread always receives a time slice before an idle-time thread.

Within each class, the scheduler maintains a priority level for a thread. The priority level can be from 0 to 31. A thread with priority level 31 always receives a time slice before a thread with level 30, and so on. If two or more threads have the same priority level, the scheduler distributes the CPU time equally among them by using a round-robin scheme; that is, the scheduler gives a time slice to first one, then another, and so on, and then goes back to the first. A process can set and retrieve the priority level by using the **DosSetPrty** and **DosGetPrty** functions.

Since the system can create and execute threads quickly, the preferred multitasking method is to distribute tasks among parts of the program instead of between programs.

A process does not have to rely on the scheduler to control execution of its threads. A process can use the **DosSuspendThread** and **DosResumeThread** functions to suspend and resume the execution of a given thread. When a process suspends a thread, the thread remains suspended until the **DosResumeThread** function is called.

A process or thread ends when it calls the **DosExit** function. MS OS/2 automatically closes any files or other resources left open by the process when it ends. When a thread ends, any resources it may have open remain open until another thread closes them or the process ends. A process can direct MS OS/2 to carry out other actions when the process ends by using the **DosExitList** function to create a list of termination functions. MS OS/2 calls the termination functions in the order given when the process is about to end.

Programs can load and execute other programs by using the **DosExecPgm** function. The new program, once loaded, is called a child process. The process that starts the new program is called the parent process. A child process is like any other process. It has its own program code and data and its own threads. The child process inherits the other resources of the parent process, such as files, pipes, and queues. The child process can use the inherited resources without preparing them. For example, if the parent process opens a file for reading and then starts the child process, the child process can read from the file immediately. It does not have to open the file for itself.

The parent process determines how the child process should run. A child process can run independently of the parent process; that is, both may run at the same time or the parent process may wait until the child process ends. A process can use the **DosCWait** function to retrieve the termination status of a child process that runs independently.

2.3 Memory-Management Functions

MS OS/2 memory-management functions let a program allocate memory for its own use or to be shared with other processes. A process can allocate a segment, a huge segment, or memory blocks within a segment.

The following are memory-management functions:

DosAllocHuge	DosGiveSeg
DosAllocSeg	DosLockSeg
DosAllocShrSeg	DosMemAvail
DosCreateCSAlias	DosReallocHuge
DosFreeSeg	DosReallocSeg
DosGetHugeShift	DosSubAlloc
DosGetSeg	DosSubFree
DosGetShrSeg	DosSubSet

A program can allocate a segment by using the **DosAllocSeg** function. A segment is any contiguous block of memory from 1 to 65,536 bytes. MS OS/2 supplies a segment selector for each segment. The selector is an unsigned integer that identifies the segment and is used to access the bytes in the segment. The selector must be combined with a segment offset to create the address of the byte to be accessed.

To create an address from a selector and an offset in a C-language program, use the following macro:

```
MAKEP(selector, offset)
```

The process can access any byte in the segment. If the process supplies an incorrect offset or attempts to access a byte outside the segment, MS OS/2 generates a protection fault and usually terminates the process after displaying a message.

The process may free a segment at any time by using the **DosFreeSeg** function. This function frees the segment and makes the memory available for allocation by other processes. After a segment is freed, the selector is no longer valid and causes a protection fault to occur if used to access memory.

The process can also change the size of a segment by reallocating it with the **DosReallocSeg** function. When a segment is reallocated, the data in the segment remains unchanged and only the size changes. If the size is increased, the new bytes are appended to the end of the segment and their value is undefined. If the size is decreased, the function removes bytes from the end of the segment.

Each process has a distinct address space; that is, other processes cannot access the segments allocated by the process. However, a process can share a segment, permitting other processes to request a selector for that segment. A process shares a segment by allocating it with the **DosAllocShrSeg** function or by specifying a shared segment when allocating it with the **DosAllocSeg** function. If **DosAllocShrSeg** is used, the process supplies a unique name for the segment; other processes can use this name with the **DosGetShrSeg** function to retrieve a segment selector. The name has the same format as an MS OS/2 filename, but no file is created for the shared segment. The name has the following form:

`\sharemem\name`

If **DosAllocSeg** is used, a process uses the **DosGetSeg** or **DosGiveSeg** function to prepare a segment selector.

A process can allocate a huge segment by using the **DosAllocHuge** function. A huge segment is actually several segments whose total memory is greater than 65,535 bytes. MS OS/2 supplies one segment selector for the huge segment. This is the selector for the first-segment. To access bytes in the other segments, the process must calculate the segment selector by adding the segment-selector offset one or more times to the first-segment selector. The selector offset is computed by using the huge-segment shift count retrieved by the **DosGetHugeShift** function. The shift count represents an exponent of 2, so a segment-selector offset equals the value 1 shifted left by the shift count. For example, the segment-selector offset is 8 if the shift count is 3. If the first-segment selector is 10, the second selector is 18, the third is 26, and so on. The process can change the size of the huge segment by using the **DosReallocHuge** function, and it can free the huge segment by using the **DosFreeSeg** function. Freeing the first segment in a huge segment frees all segments.

A process can allocate memory blocks within a segment by using the **DosSubSet**, **DosSubAlloc**, and **DosSubFree** functions. This suballocation of a segment is a way to let a process allocate small memory blocks without increasing the number of segments the system has to manage. The memory blocks are completely contained in the segment and any action the system carries out on the segment is applied to all memory blocks. A process prepares a segment for suballocation by using the **DosSubSet** function. It can then allocate or free memory blocks by using the **DosSubAlloc** and **DosSubFree** functions.

MS OS/2 can move code and data segments to make the best use of physical memory. If the **memman** command in the *config.sys* file specifies **move**, MS OS/2 consolidates free space in memory by compacting the existing code and data segments. MS OS/2 compacts memory by moving the code and data segments together so that no free memory appears between them. Compacting results in the largest contiguous block of free memory possible.

MS OS/2 can swap data segments to let programs allocate more memory than is actually available in the system. If the **memman** command in the *config.sys* file specifies **swap**, MS OS/2 writes selected data segments to the *swapper.dat* file whenever no physical memory exists for a given allocation request. MS OS/2 selects the data segments to swap based on when they were last used. If a program ever needs the segments, MS OS/2 reads them from the *swapper.dat* file into memory, possibly swapping other segments to make room.

A program can allocate discardable segments. A discardable segment is like any other segment except that when the system needs additional space, it can discard the segment; that is, it can reduce the segment's size to zero and delete the segment's data without saving it on disk. Discardable segments are intended to be used for code or data that can be quickly regenerated. While the segment is in use, the program can lock it by using the **DosLockSeg** function to prevent the segment from being discarded. When the segment is not in use, the program can unlock it and make it discardable by using the **DosUnlockSeg** function.

Although MS OS/2 can access no more than 16 megabytes of physical memory, it can manage segment selectors representing up to 1 gigabyte of memory. MS OS/2 uses compaction, discarding, and swapping to manage this "virtual" memory.

When MS OS/2 runs only protected-mode programs (the **protectonly** command in the *config.sys* file is set to **yes**), all physical memory is available to protected-mode code and data segments except the memory addresses from 640K to 1000K. These are reserved for system ROM and video buffers. When MS OS/2 runs real- and protected-mode programs (**protectonly** is set to **no**), it reserves lower memory for real-mode programs. The **rmsize** command in the *config.sys* file specifies the size and the upper address of real-mode memory. The system places any protected-mode code and data segments above the last upper real-mode address. Real-mode memory is not subject to swapping or moving unless the program running in real-mode memory does so itself.

In protected mode, the system creates one local descriptor table (LDT) for each process. All threads in that process have access to the same LDT. MS OS/2 reserves the global descriptor table (GDT) for system code and data segments.

In some cases, a process may need to create and execute code while the process is running. Although a process cannot pass execution control to code in an allocated segment, it can create a code-segment alias for a data segment by using the **DosCreateCSAlias** function. The function takes a data-segment selector and supplies a code-segment selector that can be used to call code in the segment.

2.4 Dynamic-Link Functions

MS OS/2 dynamic-link functions let a program link to a function or set of functions while the program runs. A program can load a dynamic-link library, also called a module, retrieve the address of a function in the library, and call the function to carry out a task.

The following are dynamic-link functions:

DosFreeModule
DosGetModHandle
DosGetModName
DosGetProcAddr
DosLoadModule

A program can link to any function in any dynamic-link library. First, the program must load the library by using the **DosLoadModule** function. This function takes the name of the library (without the *.dll* extension) and returns a handle that identifies the loaded library. The program then must retrieve the address of the desired function by using the **DosGetProcAddr** function. **DosGetProcAddr** takes a function name as a parameter and supplies the address. The function name must be spelled exactly as it appears in the library; this is typically all uppercase letters if the Pascal calling convention is used. The function can be called like any MS OS/2 system function. The program is responsible for providing correct parameters in correct order and type.

After a program has finished using a library, it can free the library by using the **DosFreeModule** function. If no other program is using the library, MS OS/2 removes the library from memory. The system keeps only one copy of a library in memory, no matter how many programs have loaded it.

For more information about how to create dynamic-link libraries, see *Microsoft Operating System/2 Programming Tools*.

2.5 File-System Functions

MS OS/2 file-system functions let a program open, read, write, and modify disk and device files. These functions also let a program access and maintain the file system, that is, the volumes, directories, and files on the computer's disk drives.

The following are file-system functions:

DosBufReset	DosNewSize	DosRmdir
DosChdir	DosOpen	DosSelectDisk
DosChgFilePtr	DosQCurDir	DosSetFHandState
DosClose	DosQCurDisk	DosSetFileInfo
DosDelete	DosQFHandState	DosSetFileMode
DosDupHandle	DosQFileInfo	DosSetFSInfo
DosFileLocks	DosQFileMode	DosSetMaxFH
DosFindClose	DosQFSInfo	DosSetVerify
DosFindFirst	DosQHandType	DosWrite
DosFindNext	DosQVerify	DosWriteAsync
DosMkdir	DosRead	
DosMove	DosReadAsync	

In MS OS/2, the file system specifies how data is organized on the computer's mass storage devices, such as hard and floppy disks. Each disk drive represents a unique element of the file system through which the data on a fixed disk (hard disk) or a removable disk (floppy disk) may be accessed. Each drive is given a unique letter to distinguish it from other drives. On most computers, drive A is the first floppy-disk drive, drive B is the second floppy-disk drive, drive C is the first hard-disk drive, and drive D is the second hard-disk drive. A computer may have up to 26 drives. For exceptionally large hard disks, the disk may be divided into two or more "logical" drives. A logical drive represents a portion of the hard disk (up to 32 megabytes of storage) and, like a physical drive, is given a unique letter to distinguish it from other physical and logical drives.

The file system organizes disks into volumes, directories, and files. A volume is the largest file-system unit. It represents all the available storage on the disk in the drive. An optional volume identifier or name identifies the disk. Each volume has a root directory. The root directory lists the contents of the disk. Each directory entry identifies the name, location, and size of the files and directories on the disk. A file is one or more bytes of data stored on the disk. Subdirectories provide additional storage for files and, like the root directory, contain directory entries.

A program can open any existing file in the file system, create new files or directories for the file system, and delete existing files and directories. The program opens files by using the **DosOpen** function and specifying the name of the file to be opened. The function searches for the name in the current directory on the current drive unless the name explicitly specifies the directory and drive to search. Valid MS OS/2 filenames have the following form:

[*drive:*][\ *directory-name* \] *filename* [*.filename-extension*]

The drive letter is specified in *drive* and must be followed by a colon (:); *drive* must name an existing drive, and can be any letter from A to Z.

The name of the directory that contains the file's directory entry is specified by *directory-name*. It must be followed by a backslash (\) to separate it from the *filename*; *directory-name* must name an existing directory in the current directory. If the specified directory is not in the current directory, *directory-name* must be preceded by the name of the directory that contains it. For example, if the directory "abc" is in the directory "sample" and "sample" is in the root directory, the correct *directory-name* would be \sample\abc. The root directory is specified by using a backslash (\) at the beginning of the name. A directory name consists of any combination of up to eight letters, digits, or the following special characters:

\$ % ' - _ @ { } ~ ' ! # ()

A directory name may also have an extension, which is any combination of up to three letters, digits, or special characters preceded by a period (.).

The file is named by *filename* and *filename-extension*; *filename* may be any combination of up to eight letters, digits, or the following special characters:

\$ % ' - _ @ { } ~ ' ! # ()

The extension, *filename-extension*, is preceded by a period and may be any combination of up to three letters, digits, or special characters.

Although these file-naming conventions apply to MS OS/2 1.0, future releases of MS OS/2 may use other conventions. Therefore, programs written for MS OS/2 must not depend on any specific filename format.

A program can select the current directory and drive of the file system by using the **DosChdir** and **DosSelectDisk** functions. When a program starts, it inherits the current directory and drive from the program that starts it. The **DosQCurDir** and **DosQCurDisk** functions let a program determine which directory and drive are current.

When a program creates a new file, the system adds an entry for the file to the specified directory. Each directory can have any number of entries, up to the physical limit of the disk. A program can create new directories and delete existing directories by using the **DosMkdir** and **DosRmdir** functions. To delete a directory, the program must make sure that the files and directories in that directory have also been deleted or moved. The program can delete a file by using the **DosDelete** function or move a file to another directory by using the **DosMove** function.

Each directory entry for a file or directory includes a file attribute. The file attribute defines whether the directory entry specifies a file, directory, or volume identifier. It also specifies whether the file is read-only, hidden, archived, or a system file. A read-only file cannot be opened for writing or

be deleted. A hidden file or directory cannot be displayed using an ordinary directory listing. A system file is excluded from normal directory searches. The archived attribute is for special-purpose programs that need some way to mark a file for backing up or removal. A program can retrieve and set the file attribute of a file by using the **DosQFileMode** and **DosSetFileMode** functions.

A program can retrieve information about the file system on a given drive by using the **DosQFSInfo** function. The file-system information defines the amount of storage space available on the disk. The storage space is given in number of allocation units (clusters) on the disk. Each cluster has an associated number of sectors; each sector contains a given number of bytes. A typical disk has 512 bytes for each sector and four sectors for each cluster. The **DosSetFSInfo** function lets a program change the volume identifier for the disk in the given drive.

A program can retrieve and set information about a file by using the **DosQFileInfo** and **DosSetFileInfo** functions. File information consists of the dates and times that the file was created, last accessed, and last written to. File information also specifies the size (in bytes) of the file, the number of sectors (or clusters) the file occupies, and the file attributes.

A program can search for all filenames that match a given pattern by using the **DosFindFirst**, **DosFindNext**, and **DosFindClose** functions. These functions let a program search the current directory for files whose names match a specified pattern. The pattern is an MS OS/2 filename and can include wildcard characters. The wildcard characters are the question mark (?) and the asterisk (*). The ? matches any character, the * matches any combination of characters. For example, the pattern "a*" matches the names *abc*, *a23*, and *abca*, but the pattern "a?c" matches only the name *abc*.

A program can open an existing file or create a new file by using the **DosOpen** function. MS OS/2 identifies each open file by supplying a file handle when the program opens or creates the file. The file handle is a unique 16-bit integer. The program may use the handle in functions such as **DosRead** and **DosWrite** to read from and write to the file, depending on how the file was opened. The program can continue to use the handle in this way until the file is closed by using the **DosClose** function. MS OS/2 sets the default maximum number of file handles for a program to 20. A program can change this maximum by using the **DosSetMaxFH** function.

When a program opens a file, it must specify whether it wants to read from the file, write to it, or both read and write. Also, since more than one program may attempt to open a file, a program must specify how it wants to share the file (if at all) with other programs. A file can be shared for reading only, writing only, reading and writing, or not shared. A file that is not shared cannot be opened by another program (or even the same program) until it has been closed.

A program reads from or writes to the file by using the **DosRead** and **DosWrite** functions. These functions read and write a specified number of bytes of data. The data is read and written exactly as given. The functions do not format the data; that is, they do not convert binary data to decimal strings or vice versa. A program may use the **DosReadAsync** and **DosWriteAsync** functions to read from and write to a file asynchronously; that is, it reads and writes as separate operations while the rest of the program continues carrying out other operations.

Every open file has a file pointer that specifies the next byte to be read or the location to receive the next byte that is written. When a file is first opened, the system places the file pointer at the beginning of the file. As each byte is read or written, the system advances the pointer. A program can also move the file pointer by using the **DosChgFilePtr** function. When the pointer reaches the end of the file and an attempt is made to read from the end, no bytes are read and no error occurs. Reading zero bytes without an error means the program has reached the end of the file. A program can increase or decrease the size of an open file by using the **DosNewSize** function.

When a program writes to a disk file, MS OS/2 usually collects the data being written in an internal buffer and only writes to the disk when the amount of data equals (or is a multiple of) the sector size of the disk. If there is data in the internal buffer when the file is closed, the system automatically writes the data to the disk before closing the file. A program can also flush the buffer (that is, write the contents of the buffer to the disk) by using the **DosBufReset** function.

Although MS OS/2 lets more than one program open a file and write to it, programs that do so must take care to prevent writing over the work of other programs. A program can temporarily lock a region in a file to prevent other programs from reading from it or writing to it. The **DosFileLocks** function specifies a range of bytes in the file that is locked by the given program and that may only be accessed by that program. The program uses the same function to free the locked region.

Some MS OS/2 file-system functions can also be used to access other input and output devices, such as serial ports, keyboard, and screen. A program can open these devices by using the **DosOpen** function and specifying one of the following device names:

clock\$	con	mouse\$
com1	kbd\$	nul
com2	lpt1	pointer\$
com3	lpt2	prn
com4	lpt3	screen\$

MS OS/2 supplies a unique file handle that the program may use in subsequent calls to the **DosRead** and **DosWrite** functions to read from and write to the device. A program can determine the type of handle, device, or file system by using the **DosQHandType** function. The program uses the **DosClose** function to close the handle when the device is no longer needed.

When a program opens a device, it must specify how the device is to be accessed (read, write, or both) and what kind of sharing is to be allowed. A program can check and even alter this state by using the **DosQFHandState** and **DosSetFHandState** functions. These functions also apply to file-system handles.

When a program starts, it inherits all open file handles from the process that starts it. If the system's command processor starts a program, file handles 0, 1, and 2 represent the standard input, standard output, and standard error files. The standard input file is the keyboard, the standard output and error files are the screen. A program can read from the standard input and write to the standard output and error files immediately; that is, it does not have to open the files first.

A program can create a duplicate file handle for an open file by using the **DosDupHandle** function. A duplicate handle is identical to the original handle. Typically, duplicate handles are used to redirect the standard input and output files. For example, a program can close handle 0, open a disk file, and duplicate the disk-file handle as handle 0. Thereafter, reading from handle 0 reads from the disk file, not the keyboard.

2.6 Input-and-Output Control Functions

Although a program may read from and write to a device just as if it were a file, some actions that a program may need to carry out on the device must be done by using the **DosDevIOCtl** function. This function lets a program control a specific device; that is, the function sends it device-specific commands, such as setting the baud rate for a serial device. A program sends a command by using the **DosDevIOCtl** function to call an input-and-output control function (called an IOCtl function).

MS OS/2 provides IOCtl functions for the keyboard, screen, mouse, serial ports, and disk drives. In most cases, a program should be able to use the MS OS/2 system functions to carry out input and output through these devices. Occasionally, however, a program needs an IOCtl function to carry out a special input or output task. For a full description of the IOCtl functions, see Chapter 4, "Input-and-Output Control Functions."

2.7 Keyboard Functions

MS OS/2 keyboard functions give programs direct access to the system keyboard. Programs may read individual keystrokes from a keyboard, or they may read complete strings. Keystroke information includes the character value, the scan code for the key, and the status of the keyboard, such as the state of the shift keys.

The following are keyboard functions:

KbdCharIn	KbdPeek
KbdClose	KbdRegister
KbdDeRegister	KbdSetCp
KbdFlushBuffer	KbdSetCustXt
KbdFreeFocus	KbdSetStatus
KbdGetCp	KbdStringIn
KbdGetFocus	KbdSynch
KbdGetStatus	KbdXlate
KbdOpen	

A program can read individual keystrokes by using the **KbdCharIn** function, or read a string of characters by using the **KbdStringIn** function. MS OS/2 stores each keystroke in an input buffer for the keyboard. The **KbdCharIn** and **KbdStringIn** functions read the next keystroke from the buffer and copy it to the structure or buffer specified by the program. A program can look at the next character in the buffer without removing it by using the **KbdPeek** function, or flush the contents of the buffer by using the **KbdFlushBuffer** function.

The keyboard functions require a keyboard handle that identifies the keyboard to read from or modify. A program can always use keyboard handle 0 to read from the system keyboard. However, if more than one process (or thread) in the same screen group attempts to read from the system keyboard at the same time, there is no guarantee the correct process will receive the keystrokes intended for it. To prevent these conflicts, a program can create a logical keyboard by the **KbdOpen** function. A logical keyboard receives no keyboard input until it is given the keyboard focus by the **KbdGetFocus** function. Only the logical keyboard that has the focus can receive input. A process relinquishes the keyboard focus by using the **KbdFreeFocus** function. The **KbdClose** function closes the logical keyboard.

A program can change the status of a keyboard by using the **KbdSetStatus** function. The keyboard status defines how the keyboard operates. Two important features of the status are echo mode and input mode. Echo mode defines whether or not characters are displayed on the screen as they are typed. Echo mode may be either on or off. Input mode defines whether MS OS/2 control and editing keys are processed when characters are typed. Input mode may be cooked or raw. In cooked mode, all MS OS/2

control and editing keys, such as CONTROL+C and F3, are processed when the program reads characters by using the **KbdStringIn** function, and all MS OS/2 control keys are processed when the program reads key-strokes by using the **KbdCharIn** function. In raw mode, only the CONTROL+BREAK control key is processed in both cases.

The following key combinations are MS OS/2 control keys:

CONTROL+C	CONTROL+S
CONTROL+H	CONTROL+Z
CONTROL+J	CONTROL+BREAK
CONTROL+P	

The following are MS OS/2 editing keys:

F1	DELETE
F2	ESCAPE
F3	INSERT
F4	BACKSPACE
F5	

Other keyboard functions carry out special tasks for adapting the keyboard to receive country-specific information or for modifying the operation of one or more of the keyboard functions.

2.8 Mouse Functions

MS OS/2 mouse functions give programs direct access to the system mouse or other pointing device. Programs may read individual events from the mouse and carry out actions based on those events. Mouse events are mouse motions and button presses.

The following are mouse functions:

MouClose	MouGetNumQueEl	MouSetEventMask
MouDeRegister	MouGetPtrPos	MouSetHotKey
MouDrawPtr	MouGetPtrShape	MouSetPtrPos
MouFlushQue	MouGetScaleFact	MouSetPtrShape
MouGetDevStatus	MouInitReal	MouSetScaleFact
MouGetEventMask	MouOpen	MouSetDevStatus
MouGetHotKey	MouReadEventQue	MouSynch
MouGetNumButtons	MouRegister	
MouGetNumMickeys	MouRemovePtr	

A program reads mouse events by first opening the mouse by using the **MouOpen** function, and then drawing the mouse pointer by using the **MouDrawPtr** function. MS OS/2 copies mouse-event information to the

mouse queue. The program retrieves mouse events by using the **MouReadEventQue** function. The event defines the position of the mouse and the state of the mouse buttons. The program may move the mouse pointer by using the **MouSetPtrPos** function or retrieve the current position by using the **MouGetPtrPos** function. If a program needs to temporarily hide the mouse pointer, the **MouRemovePtr** function removes the mouse pointer from all or a portion of the screen. When a program has finished using the mouse, it can close it by using the **MouClose** function.

MS OS/2 specifies the position of the mouse in either screen coordinates or as the number of mickeys relative to the last position. Screen coordinates are relative to the upper-left corner of the screen. The x -axis values increase to the left, the y -axis values increase downward. The screen units are either character cells or pixels, depending on the screen mode (character cells for text mode, pixels for graphics mode). The position never exceeds the width or height of the screen.

If the mouse position is specified in mickeys, it represents the direction and distance the mouse has moved since its last position. A mickey is the mouse unit of motion. A program can determine how many centimeters the mouse has moved by using the **MouGetNumMickeys** function to retrieve the mouse's mickey-to-centimeter ratio. A position in mickeys is a signed value. If the x -coordinate is negative, the mouse moved to the left; if positive, it moved to the right. If the y -coordinate is negative, the mouse moved up; if positive, it moved down.

MS OS/2 reports only the mouse events that are defined in the mouse event mask. A program can set the event mask by using the **MouSetEventMask** function or retrieve the event mask by using the **MouGetEventMask** function. The program can determine how many mouse events are in the queue by using the **MouGetNumQueEl** function and flush any existing events by using the **MouFlushQue** function.

A program can change the shape of the mouse pointer by using the **MouSetPtrShape** function. Typically, a program first retrieves the current shape by using the **MouGetPtrShape** function, and then modifies that shape before setting it by using the **MouSetPtrShape** function. The mouse pointer shape consists of two masks: an AND mask and an XOR mask. MS OS/2 first combines the AND mask with the contents of the screen at the current position by using the bitwise AND operator. It then combines the result with the XOR mask by using the bitwise XOR operator. The format of the masks depends on the screen mode. They are character-and-attribute pairs for text mode, and bitmaps for graphics mode.

MS OS/2 automatically updates the mouse pointer position as the mouse moves, but only does so if the amount of motion is greater than or equal to the scaling factor. The scaling factor, set by using the **MouSetScaleFact** function, defines the number of mickeys the mouse must move before the

mouse pointer moves one screen unit. For example, in text mode, the horizontal and vertical scaling factors are usually the same as the width and height of a character cell. This means the mouse moves one character cell at a time rather than moving within a cell.

MS OS/2 cannot draw the mouse pointer unless a pointer driver has been specified in a **device** command in the *config.sys* file, or a pointer driver name is explicitly given when the **MouOpen** function is used. The pointer driver supplied with the system is named *pointdd.sys*.

Other mouse functions carry out special tasks for adapting the mouse for real-mode operation and for modifying the operation of one or more of the mouse functions.

2.9 Video I/O Functions

MS OS/2 video input/output (I/O) functions give programs direct access to the system display. Programs may write individual characters and strings to the screen. Strings can consist of characters, attributes, or character-and-attribute pairs.

The following are screen functions:

VioDeRegister	VioPrtSc	VioSetCurPos
VioEndPopUp	VioPrtScToggle	VioSetCurType
VioGetAnsi	VioReadCellStr	VioSetFont
VioGetBuf	VioReadCharStr	VioSetMode
VioGetConfig	VioRegister	VioSetState
VioGetCp	VioSavRedrawUndo	VioShowBuf
VioGetCurPos	VioSavRedrawWait	VioWrtCellStr
VioGetCurType	VioScrLock	VioWrtCharStr
VioGetFont	VioScrollDn	VioWrtCharStrAtt
VioGetMode	VioScrollLf	VioWrtNAttr
VioGetPhysBuf	VioScrollRt	VioWrtNCell
VioGetState	VioScrollUp	VioWrtNChar
VioModeUndo	VioScrUnLock	VioWrtTTY
VioModeWait	VioSetAnsi	
VioPopUp	VioSetCp	

A program can write characters to the screen by using the **VioWrtCharStr** and **VioWrtNChar** functions. These functions write a string of one or more characters to the specified position. The program can also write a string of attributes to the screen by using the **VioWrtCharStrAtt** and **VioWrtNAttr** functions. Attributes define the color and appearance of the characters at the corresponding position. The functions

write the attributes to the specified position. The effect of a given attribute depends on the display. For a list of attribute values and meanings for some common displays, see Appendix B, “Devices.”

A program can combine character and attribute values into a single value, called a cell or a character-and-attribute pair, and write one or more cells to the screen by using the **VioWrtCellStr** and **VioWrtNCell** functions. These functions write both the character and attribute to the specified position.

The simplest screen-output function is **VioWrtTTY**. It writes a given string of characters to the screen starting at the current cursor position. It advances the cursor to the right as it writes each character. **VioWrtTTY** is similar to the **DosWrite** function when it is used to write strings to the standard output.

A program can set the position of the cursor or retrieve its current position by using the **VioSetCurPos** and **VioGetCurPos** functions. The cursor position, like any position on the screen, is specified in screen coordinates. Screen coordinates are relative to the upper-left corner of the screen. The *x*-axis values increase to the right, the *y*-axis values increase downward. The screen units are either character cells or pixels, depending on the screen mode (character cells for text mode, pixels for graphics mode). The position never exceeds the width or height of the screen. The program can set the screen mode by using the **VioSetMode** function or retrieve the screen mode by using the **VioGetMode** function. The type of modes that can be set depends entirely on the display. For more information, see Appendix B, “Devices.” A program can also set the cursor type by using the **VioSetCurType** function. The cursor type defines the width and height of the cursor as well as its color and appearance.

A program can scroll the screen to the left, right, up, or down by using the **VioScrollLf**, **VioScrollRt**, **VioScrollUp**, and **VioScrollDn** functions. These functions move the specified rectangle to the given location on the screen, filling any area uncovered by the rectangle with the given character and attribute.

A character is actually specified by a character value. The screen uses the value to locate a character bitmap in the current screen font and it displays the bitmap at the character's location. For some displays, a program can change the current screen font by using the **VioSetFont** function. Typically, a program first retrieves the screen font by using the **VioGetFont** function, and then modifies that font before setting it as the new font. The format of the font bitmap depends on the display.

Other screen functions carry out special tasks for adapting the screen for country-specific or ANSI information, for modifying the operation of one or more of the screen functions, and for accessing the video buffers directly.

2.10 Pipe Functions

The MS OS/2 pipe function, **DosMakePipe**, lets a program create a pipe that can be used to transfer information between processes. A pipe is a special internal file a process can write to and read from. The **DosMakePipe** function creates the pipe and supplies file handles to the pipe. One handle permits writing to the pipe, the other permits reading. A process can write to the pipe by using the **DosWrite** function. It can read from the pipe by using the **DosRead** function.

A pipe is typically used to direct the output of one process to standard input of another. To do this, a process opens a pipe, duplicates the pipe read handle as the standard input file for a child process, and then starts the child process. The parent process can then write to the pipe and the child process can read what the parent process has written.

A pipe continues until both handles are closed. There can be no more than 65,535 bytes of unread data in a pipe at any given time. The **DosWrite** function may wait for data to be read from the pipe before completing its operation. If the read handle is closed before the write handle is closed, writing to the pipe generates an error.

2.11 Queue Functions

MS OS/2 queue functions let a program create a queue that can be used to receive information from other processes. Processes pass information through a queue in the form of messages. A process writes a message to the queue; the process that owns the queue can then read the message from the queue.

The following are queue functions:

DosCloseQueue	DosPurgeQueue
DosCreateQueue	DosQueryQueue
DosOpenQueue	DosReadQueue
DosPeekQueue	DosWriteQueue

A program creates a queue by using the **DosCreateQueue** function and specifying a unique queue name. The queue name is similar to an MS OS/2 file name, but no actual file is created for the queue. The queue name has the following form:

\queues\name

Once the queue has been created, other processes can open the queue by using the **DosOpenQueue** function and supplying the specified queue name. Processes that open the queue can write messages to it by using the **DosWriteQueue** function. Only the process that created the queue can read messages from it by using the **DosReadQueue** function. The owning process can also examine messages without removing them by using the **DosPeekQueue** function, or remove all messages from the queue by using the **DosPurgeQueue** function.

The format of a queue message depends entirely on the process creating the queue. The format and content must be understood by the process writing messages to the queue. The system automatically supplies the process identifier of the process that adds a message to the queue, so that the owning process can determine the origin of the message.

The order in which the owning process reads messages from the queue depends on the type of queue. A queue can have first-in/first-out, last-in/first-out, or priority ordering. In priority ordering, the message with the highest priority is read first. Priority values range from 0 to 15.

2.12 Semaphore Functions

MS OS/2 semaphore functions let a program create and use system and RAM semaphores. A semaphore is a special variable that a program can use to signal the beginning and ending of a given operation. Semaphores are typically used in conjunction with a limited resource to prevent more than one process or thread within a process from accessing the resource at the same time.

The following are semaphore functions:

DosCloseSem	DosSemRequest
DosCreateSem	DosSemSet
DosMuxSemWait	DosSemSetWait
DosOpenSem	DosSemWait
DosSemClear	

A process can create a semaphore in one of two ways: If the process intends to share the semaphore with other processes, it must create a system semaphore by using the **DosCreateSem** function. If the process intends to use the semaphore only within the threads of the process, it can use a RAM semaphore. A RAM semaphore is an unsigned long variable defined as a global variable for the program. To use it correctly, the variable must be initialized to zero.

MS OS/2 supplies a semaphore handle when a system semaphore is created. The process can use this handle in subsequent semaphore functions to set, clear, and wait for the semaphore. To use a RAM semaphore, the process simply passes a pointer to the unsigned long variable in these functions.

A system semaphore has a unique name that the process creating the semaphore must supply. The semaphore name is like an MS OS/2 filename, but no actual file is created for the queue. The semaphore name has the following form:

`\sem\name`

If any other process wants to use the system semaphore, it can open the semaphore by using the **DosOpenSem** function and supplying the specified semaphore name. When a process is finished using a system semaphore, it can close the semaphore by using the **DosSemClose** function.

A process can set or clear a semaphore by using the **DosSemSet** and **DosSemClear** functions. If a system semaphore is an exclusive semaphore (as specified when created), only the process that created the semaphore may set or clear it. Once a semaphore is set, the process can wait for that semaphore to become clear by using the **DosSemWait** function. The function does not return control until the semaphore is cleared. Ideally, one process or thread waits for the semaphore while another carries out a task and then clears the semaphore. A process can also wait for a given semaphore to become clear by using the **DosSemSetWait**, **DosSemRequest**, and **DosMuxSemWait** functions.

2.13 Signal Functions

MS OS/2 signal functions let a program receive and process the system signals. A system signal is special input from the user or another process that causes the process to temporarily suspend execution while the signal-handler function is executed.

The following are signal functions:

DosFlagProcess
DosHoldSignal
DosKillProcess
DosSendSignal
DosSetSigHandler

A process receives a signal whenever the user presses the CONTROL+C or CONTROL+BREAK keys while the process is running. A process also receives a signal when another process uses the **DosSendSignal**, **DosFlagProcess**, or **DosKillProcess** function to send a signal. For CONTROL+C and CONTROL+BREAK, the current foreground process or the last process to use the **DosSetSigHandler** function receives the signal. Otherwise, the specified process receives the signal. When a process receives a signal, MS OS/2 suspends execution of the main thread (thread 1) of the process and passes control to the signal handler of the process. If a process has not explicitly set a signal handler by using the **DosSetSigHandler** function, a default signal handler is used. The default signal handlers for the CONTROL+C, CONTROL+BREAK, and **DosKillProcess** signals terminate the process. The signal handlers for the flag signals ignore the signal.

A process can replace the default signal handler for any signal by using the **DosSetSigHandler** function. A signal handler is simply a function that receives control when the signal occurs. Although it has a specific form, it can carry out any action, such as cleaning up and saving files before terminating the process. The handler can carry out any action and either terminate the program or return control to the point at which the process was suspended.

2.14 Timer Functions

MS OS/2 timer functions let a program time events. These functions take an interval of time specified in milliseconds and either wait for the interval to elapse or clear a semaphore after the interval has elapsed.

The following are timer functions:

DosSleep
DosTimerAsync
DosTimerStart
DosTimerStop

A process can wait for a given interval of time to elapse by using the **DosSleep** function. If the process wants to carry out other tasks while the timer counts out the interval, it can use the **DosTimerAsync** function. This function clears a given semaphore when the specified time has elapsed. If the process wants the timer to continue to count out the intervals, it can use the **DosTimerStart** function. Unlike the **DosTimerAsync** function, **DosTimerStart** does not stop after the first interval is counted. It repeats the count and clears the semaphore each time the interval elapses. A process can stop a timer by using the **DosTimerStop** function.

The timers rely on the system clock to compute the elapsed time. The system clock keeps a count of the number of system-clock interrupts that have occurred since the system was started. System-clock interrupts occur approximately 32 times a second, so timer intervals that are less than 50 milliseconds are not recommended. All time values are in milliseconds and are rounded up to the next clock tick. The duration of the clock tick can be determined by using the **DosGetInfoSeg** function and examining the timer-interval field in the global information segment. The global information segment also contains the current system time in milliseconds. The system time is the number of milliseconds that have elapsed since the system started. A process that needs to know precise time between the start and end of a timer can save the system time before starting the timer and compare it with the system time after the timer ends. The system time is reset to zero every few weeks, so a process may need to take this into account when comparing starting and ending times. The system time may occasionally lose a millisecond if a process disables interrupts for periods longer than the clock-tick interval. However, the time of day (hours, minutes, seconds), the time in seconds since 1-1-70, and the date will remain accurate.

2.15 Device-Monitor Functions

MS OS/2 device-monitor functions examine and modify input from devices such as the keyboard and mouse before input is available to any other program. A device monitor is a special type of program input in which a program receives raw input directly from a device such as the keyboard or mouse before the input is passed on to any program that may be reading from the device. A program can use a device monitor to inspect, modify, insert, and remove information as it passes from the device to other programs.

The following are device-monitor functions:

DosMonClose
DosMonOpen
DosMonRead
DosMonReg
DosMonWrite

A program creates a device monitor for a given device by using the **DosMonOpen** function. The program does not receive input from the device until it uses the **DosMonReg** function to register the input and output buffers to be used with the monitor. These buffers are structures

into which MS OS/2 copies the input or reads the output. The format of the structures depends on the device. Typically, a program retrieves the correct size of the input and output buffers by trying to register the buffers with an incorrect size. The **DosMonReg** function copies the correct size to the first field in each structure.

Once a monitor is registered, a program can retrieve input from the device by using the **DosMonRead** function. It can modify this information and pass it on by using the **DosMonWrite** function. If it does not pass the information on by using the **DosMonWrite** function, that information is lost.

2.16 Error-Handling Functions

MS OS/2 error-handling functions let a program get additional information about error codes returned by the MS OS/2 functions. They also let the program control the default error processing performed by the system when hard errors occur.

The following are error-handling functions:

DosErrClass
DosError
DosSetVec
DosSystemService

Every MS OS/2 system function returns an error code if the function is not able to carry out the requested task. A program can retrieve more information about the error, such as the reason for the error and the recommended course of action, by using the **DosErrorClass** function. This information is useful for determining if the error can be resolved using software or if it is a hardware problem.

In MS OS/2, a hard error is any error a program cannot resolve using software. For example, having the disk-drive door open when a program tries to read from a file on that drive is a hard error; encountering a division by zero exception is also a hard error. When a hard error occurs, the system typically passes control to the MS OS/2 session manager, which displays an error message and prompts for user input.

A program can override this default processing by using the **DosError** function to disable it or by using the **DosSetVec** function to direct exception processing to the program's own error-handling routines. When error handling is disabled, any function that encounters a hard error returns

immediately with an error code. It is then up to the program to determine the cause of the error, possibly by using the **DosErrorClass** function. When a program has set its own exception-handling routine, the system passes control to the routine. The routine, typically written in assembly language, can carry out any desired processing and either return control to the program or terminate it.

Although any process can use the **DosSystemService** function to take responsibility for default hard-error handling, this function should be reserved for use only by the session manager.

2.17 Program-Startup Functions

MS OS/2 program-startup functions give a program access to its environment segment as well as information about the version of MS OS/2. The environment segment contains a copy of the command line used to start the program and the definitions of all the environment variables available to the program.

The following are program-startup functions:

DosGetEnv
DosGetVersion
DosScanEnv
DosSearchPath

Unlike MS-DOS, MS OS/2 does not prepare a program segment prefix (PSP) for protected-mode programs. Instead, it creates an environment segment that contains the definitions for environment variables and the program command line. A program can retrieve definitions from this environment segment by first using the **DosGetEnv** function to retrieve the segment selector. The **DosScanEnv** and **DosSearchPath** functions also permit the program to use the information in the environment segment.

When MS OS/2 first starts a program, it sets the CPU registers to the following values:

Value	Description
cs:ip	Contains the program's initial entry point. This is the same as the entry point specified in the executable file.
ss:sp	Contains the starting address of the stack. This is the same stack address as specified in the executable file.

ds	Contains the segment selector of the automatic data segment. The automatic data segment is specified in the executable file.
es	Contains zero.
ax	Contains the segment selector of the environment segment. The environment segment contains the environment strings and command-line arguments for the program.
bx	Contains the offset to the start of the program command line from the beginning of the environment segment.
cx	Contains the size in bytes of the automatic data segment. If it is zero, the segment is 65,536 bytes.
bp	Contains zero.

A program written in assembly language can use these registers to access the command line and environment string. Programs written in high-level languages can retrieve the environment segment selector and command-line offset by using the **DosGetEnv** function.

When MS OS/2 starts a dynamic-link library, it sets the CPU registers to the following values:

Value	Description
cs:ip	Contains the entry-point address of the library initialization function.
ss:sp	Contains the address of the current system stack. The initialization function may use the stack to define local variables and call other functions, but it must restore the stack to its previous state before returning.
ds	Contains the segment selector of the library's automatic data segment (if it has one), or contains the selector for the data-segment program or system library that called the DosLoadModule function to load the library.
ax	Contains the module handle of the dynamic-link library.

All other register values are undefined.

The initialization function can carry out any task, but must return using an inter-segment return instruction. All registers, except **ax**, must be restored to their previous values. The function can use the **ax** register to indicate whether it was successful. The system expects the function to set the **ax** register to a nonzero value if the function was successful, or zero if the function failed.

2.18 Miscellaneous Functions

There are several additional MS OS/2 functions that let programs carry out miscellaneous tasks. These functions create and display messages, trace program execution, create and manage sessions, carry out special input and output operations, and manage language- and country-specific information.

The following are miscellaneous functions:

DosBeep	DosGetDateTime	DosPutMessage
DosCaseMap	DosGetDBCSEv	DosSelectSession
DosCLIAccess	DosGetInfoSeg	DosSetCp
DosDevConfig	DosGetMachineMode	DosSetDateTime
DosEnterCritSec	DosGetMessage	DosSetSession
DosExitCritSec	DosInsMessage	DosStartSession
DosGetCollate	DosPhysicalDisk	DosStopSession
DosGetCp	DosPortAccess	
DosGetCtryInfo	DosPTrace	

2.19 Family Application Programming Interface

Many MS OS/2 functions can also be used in programs intended to be run in real mode. These functions, collectively called the family application programming interface (family API), let developers create MS OS/2 programs that can run in both protected and real modes; that is, they can run under MS OS/2 and under MS-DOS 2.x and 3.x.

To use the family API in real-mode programs, you must use only the MS OS/2 functions that belong to the family API and observe the restrictions that apply to these functions when running in real mode. Also, you must bind your program by using the **bind** program. The **bind** program supplies the code needed to link the MS OS/2 functions to the corresponding MS-DOS system calls. This code is used only when the program is run in real mode; that is, a bound program can still run in protected mode.

Not all MS OS/2 functions belong to the family API and some that do belong have slightly different behavior when used in real mode than when used in protected mode. The following is a complete list of the family API

functions. Those marked with a dagger (†) operate differently in real mode than in protected mode. All other family API functions operate identically in both protected and real modes:

DosAllocHuge †	DosInsMessage †	KbdFlushBuffer †
DosAllocSeg †	DosMkdir	KbdGetStatus †
DosBeep	DosMove	KbdPeek †
DosBufReset	DosNewSize	KbdSetStatus †
DosCaseMap †	DosOpen †	KbdStringIn †
DosChdir	DosPutMessage †	VioGetBuf
DosChgFilePtr	DosQCurDir	VioGetCurPos
DosClose	DosQCurDisk	VioGetCurType
DosCreateCSAlias †	DosQFHandState	VioGetMode
DosDelete	DosQFileInfo	VioGetPhysBuf
DosDevConfig	DosQFileMode	VioReadCellStr
DosDevIOCtl †	DosQFSInfo	VioReadCharStr
DosDupHandle	DosQVerify	VioScrLock †
DosErrClass	DosRead †	VioScrollDn
DosError †	DosReallocHuge †	VioScrollLf
DosExecPgm †	DosReallocSeg †	VioScrollRt
DosExit †	DosRmdir	VioScrollUp
DosFileLocks	DosSelectDisk	VioScrUnLock
DosFindClose †	DosSetDateTime	VioSetCurPos
DosFindFirst †	DosSetFHandState †	VioSetCurType
DosFindNext †	DosSetFileInfo	VioSetMode
DosFreeSeg †	DosSetFileMode	VioShowBuf
DosGetCollate †	DosSetFSInfo	VioWrtCellStr
DosGetCtryInfo †	DosSetSigHandler †	VioWrtCharStr
DosGetDBCSEv †	DosSetVec †	VioWrtCharStrAtt
DosGetDateTime	DosSetVerify	VioWrtNAttr
DosGetEnv	DosSleep	VioWrtNCell
DosGetHugeShift	DosSubAlloc	VioWrtNChar
DosGetMachineMode	DosSubFree	VioWrtTTy
DosGetMessage †	DosSubSet	
DosGetVersion	DosWrite	
DosHoldSignal †	KbdCharIn †	

The **DosGetMachineMode** function is especially useful in family API programs since it specifies which environment the program is running in: MS OS/2 or MS-DOS.

The following list describes the differences in operation between real and protected mode of the family API functions marked with daggers (†) in the previous list:

Function	Real-Mode Restrictions
DosAllocHuge	This function rounds up the <i>usPartialSeg</i> parameter value to the next paragraph (16-byte) value. It copies the actual segment address to the variable pointed to by the <i>psel</i> parameter, not a selector.
DosAllocSeg	This function rounds up the <i>usSize</i> parameter value to the next paragraph (16-byte) value. It copies the actual segment address to the variable pointed to by the <i>psel</i> parameter, not a selector.
DosCaseMap	There is no method of identifying the boot drive in real mode. The system assumes that the <i>country.sys</i> file is in the root directory of the current drive.
DosCreateCSAlias	The selector returned is the actual segment address of the allocated memory. Freeing either the returned selector or the original selector immediately frees the block of memory.
DosDevIOCtl	<p>Not all input-and-output control functions can be used in real mode. Also, 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 in real mode. The following input-and-output control functions may be used in family API programs:</p> <p>BLOCKREMOVABLE GETFRAMECTL GETINFINITERETRY GETLOGICALMAP GETPRINTERSTATUS INITPRINTER LOCKDRIVE * REDETERMINEMEDIA * SETBAUDRATE SETFRAMECTL (for IBM Graphics Printers only) SETINFINITERETRY (applies to the current program only) SETLINECTRL SETLOGICALMAP UNLOCKDRIVE *</p> <p>* These input-and-output control functions can be used only with MS-DOS 3.2 and later versions.</p>

DosError	If the <i>fEnable</i> parameter is 0x0000, all subsequent int 24h requests fail until a subsequent call is made to the DosError function with <i>fEnable</i> set to 0x0001.
DosExecPgm	The only value allowed for the <i>fExecFlags</i> parameter is EXEC_SYNC. Other values cause errors. The buffer pointed to by the <i>pchFailName</i> parameter is filled with blanks, even if the function fails. The codeResult field of the RESULTCODES structure receives the exit code for the DosExit function or the MS-DOS call that terminates the program.
DosExit	The function exits from the currently executing program since there are no threads in the real-mode environment. If the <i>fTerminate</i> parameter is EXIT_THREAD, the entire process, not just a thread, ends.
DosFindClose	After closing a directory handle, any attempt to use the handle in a subsequent call to the DosFindNext function causes that function to return an error code. The handle must be re-opened by using the DosFindFirst function.
DosFindFirst	The <i>phdir</i> parameter must be 0x0001. Subsequent calls to this function must have a <i>phdir</i> parameter of 0x0001 unless the DosFindClose function has been called. In that case, 0x0001 or 0xffff are allowed.
DosFindNext	The <i>hdir</i> parameter must be 0x0001.
DosFreeSeg	A code-segment selector (created by using the DosCreateCSAlias function) and the corresponding data-segment selector are not unique. Freeing one frees both.
DosGetCollate	There is no method of identifying the boot drive in real mode. The system assumes that the <i>country.sys</i> file is in the root directory of the current drive.
DosGetCtryInfo	There is no method of identifying the boot drive in real mode. The system assumes that the <i>country.sys</i> file is in the root directory of the current drive.

DosGetDBCSEv	There is no method of identifying the boot drive in real mode. The system assumes that the <i>country.sys</i> file is in the root directory of the current drive.
DosGetMessage	There is no method of identifying the boot drive in real mode. The system assumes that the message file is in the root directory of the current drive.
DosHoldSignal	Only the signal-interrupt (SIGINTR) and signal-break (SIGBREAK) signals are recognized.
DosInsMessage	There is no method of identifying the boot drive in real mode. The system assumes that the message file is in the root directory of the current drive.
DosOpen	There are restrictions on the values that may be used with the <i>OpenMode</i> parameter. The parameter can be a combination of the following values:

Value	Meaning
0x0000	Read-only access mode.
0x0001	Write-only access mode.
0x0002	Read/write access mode.
0x0010	Deny read/write share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0020	Deny-write share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0030	Deny-read share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0040	Deny-none share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.

- 0x0080 Inheritance flag. Not available in MS-DOS 2.x.
- 0x4000 Write-through flag. Not available in MS-DOS 2.x.
- 0x8000 Direct-access-storage-device (DASD) flag.

The fail-on-errors flag (0x2000) is not available for real-mode programs.

- DosPutMessage** There is no method of identifying the boot drive in real mode. The system assumes that the message file is in the root directory of the current drive.
- DosRead** This function uses the **KbdStringIn** function whenever the specified file handle identifies the keyboard device. In real mode, **KbdStringIn** reads only the number of characters specified in the call, then beeps to signal the user that no additional characters can be entered. In protected mode, the user can enter characters until the keyboard buffer is full.
- DosReallocHuge** This function rounds up the *usPartialSize* parameter value to the next paragraph (16-byte) value.
- DosReallocSeg** This function rounds up the *usNewSize* parameter value to the next paragraph (16-byte) value.
- DosSetFHandState** The fail-on-errors and write-through flags (0x2000 and 0x4000) must not be set. Also, the inheritance flag (0x0080) must not be set in MS-DOS 2.x.
- DosSetSigHandler** Only the signal-interrupt (SIGINTR) and signal-break (SIGBREAK) signals are available, therefore **DosSetSigHandler** may be used to install signal handlers for only these signals. Furthermore, the SIGINTR and SIGBREAK signals are treated as the same signal, so the function accepts only the SIG_CTRLC value when setting a signal handler.
- DosSetVec** The *usVecNum* parameter may not be set to 0x0007 since this exception is not raised in machines using the 8088 or 8086 microprocessor.

KbdCharIn	The function does not copy the system time to the KBDKEYINFO structure and there is no interim character support. It retrieves characters only from the default keyboard (handle 0). The fbStatus field may be 0x0000 or 0x0001. The <i>hkbd</i> parameter is ignored.
KbdFlushBuffer	The <i>hkbd</i> parameter is ignored.
KbdGetStatus	This function does not support the interim character or the turnaround character.
KbdPeek	This function does not copy the system time to the KBDKEYINFO structure and there is no interim character support. It retrieves characters only from the default keyboard (handle 0). The fbStatus field may be 0x0000 or 0x0001. The <i>hkbd</i> parameter is ignored.
KbdSetStatus	This function does not support the interim character or the turnaround character. Raw input mode with echo mode on is not supported. The <i>hkbd</i> parameter is ignored.
KbdStringIn	The <i>hkbd</i> parameter is ignored.
VioScrLock	This function always indicates that the lock was successful.

Chapter 3

Functions Directory

3.1	Introduction	47
3.2	Functions	47

3.1 Introduction

This chapter contains an alphabetical listing of MS OS/2 functions. Many simple program examples are given to illustrate how a particular function may be used.

3.2 Functions

This section lists MS OS/2 system functions. Each function's purpose, syntax, parameters, and possible return values are given.

■ **USHORT DosAllocHuge**(*usNumSeg*, *usPartialSeg*, *psel*, *usMaxNumSeg*, *fAlloc*)
USHORT *usNumSeg*; number of 65,536-byte segments
USHORT *usPartialSeg*; number of bytes in last segment
SEL *psel*; pointer to variable for selector allocated
USHORT *usMaxNumSeg*; max. number of 65,536-byte segments
USHORT *fAlloc*; alloc/discardable flags

The **DosAllocHuge** function allocates a huge memory block. A huge memory block consists of one or more memory segments, each containing 65,536 bytes, and one additional segment having *usPartialSeg* bytes. If the *usPartialSeg* parameter is zero, no additional segment is allocated.

The **DosAllocHuge** function allocates the segments and copies the selector of the first segment to the variable specified by the *psel* parameter. Selectors for the remaining segments are consecutive and must be computed by using the selector offset. The segments can be shared or discardable segments. The process creating the segments can then share them with other processes by using the **DosGiveSeg** function.

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

Parameter	Description
<i>usNumSeg</i>	Specifies the number of 65,536-byte segments to be allocated.
<i>usPartialSeg</i>	Specifies the number of bytes in the last segment. It can be any value from 0 to 65,535. If it is zero, no additional segment is allocated.
<i>psel</i>	Points to the variable that receives the selector of the first segment.

usMaxNumSeg Specifies the reallocation maximum. This is 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.

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

Value	Meaning
0x0000	Create a non-sharable, non-discardable segment.
SEG_ GIVEABLE	Create a shared segment that the owning process can give to other processes by using the DosGiveSeg function.
SEG_ GETTABLE	Create a shared segment that other processes can retrieve by using the DosGetSeg function.
SEG_ DISCARDABLE	Create a discardable segment.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 (that is, $1 \ll 2$ is equal to 4). 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 may compute the selectors at any time and save them; they will remain available for use as long as the memory remains allocated.

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

Family API 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 into a far pointer that can access all of the memory allocated:

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

See Also

DosFreeSeg, **DosGetHugeShift**, **DosGetSeg**, **DosGiveSeg**,
DosReallocHuge

■ **USHORT** **DosAllocSeg**(*usSize*, *psel*, *fAlloc*)
USHORT *usSize*; number of bytes requested
PSEL *psel*; Receives selector allocated
USHORT *fAlloc*; allocation 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 to 65,536 bytes, as specified by the *usSize* parameter.

When **DosAllocSeg** allocates a segment, it applies the segment attributes specified by the *fAlloc* parameter. A segment can be shared or discardable. A shared segment is available to the process that creates it and to other processes. A process shares a shared segment by using the **DosGiveSeg** and **DosGetSeg** functions. A discardable segment is a segment that the system can discard, if necessary, to satisfy the allocation request of some other process. Discardable segments are useful for holding temporary data that can be quickly regenerated. 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.

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

Parameter	Description										
<i>usSize</i>	Specifies the number of bytes to be allocated. It can be any value from 0 to 65,535. If it is 0, the function allocates 65,536 bytes.										
<i>psel</i>	Points to the variable that receives the segment selector.										
<i>fAlloc</i>	Specifies the allocation flags for the segment. It can be any combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Create a non-sharable, non-discardable segment.</td></tr><tr><td>SEG_ GIVEABLE</td><td>Create a shared segment that the owning process can give to other processes by using the DosGiveSeg function.</td></tr><tr><td>SEG_ GETTABLE</td><td>Create a shared segment that other processes can retrieve by using the DosGetSeg function.</td></tr><tr><td>SEG_ DISCARDABLE</td><td>Create a discardable segment.</td></tr></table>	Value	Meaning	0x0000	Create a non-sharable, non-discardable segment.	SEG_ GIVEABLE	Create a shared segment that the owning process can give to other processes by using the DosGiveSeg function.	SEG_ GETTABLE	Create a shared segment that other processes can retrieve by using the DosGetSeg function.	SEG_ DISCARDABLE	Create a discardable segment.
Value	Meaning										
0x0000	Create a non-sharable, non-discardable segment.										
SEG_ GIVEABLE	Create a shared segment that the owning process can give to other processes by using the DosGiveSeg function.										
SEG_ GETTABLE	Create a shared segment that other processes can retrieve by using the DosGetSeg function.										
SEG_ DISCARDABLE	Create a discardable segment.										

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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, so you may compute the selectors at any time and save them; they will remain available for use as long as the memory remains allocated.

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. If a segment is discardable, the memory manager can discard the segment any time the system is low on memory. Although the data in the segment is lost, the segment can be restored to its original size by using the **DosReallocSeg** function.

To prevent the memory manager from discarding a segment, the **DosLockSeg** function can be used to lock the segment. A locked segment cannot be discarded. The **DosUnlockSeg** function unlocks the segment and permits discarding. The **DosAllocSeg** and **DosReallocSeg** functions automatically lock the segment.

The **DosFreeSeg** function frees the segment.

Family API 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 into a far pointer that can access the allocated bytes:

```
PCH pchBuffer;
SEL sel;

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

See Also

DosFreeSeg, DosGetSeg, DosGiveSeg, DosLockSeg,
DosReallocSeg, DosUnlockSeg

- **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 to 65,536 bytes, as specified by the *usSize* parameter.

A shared segment can be accessed by any process that knows the segment's 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 are not the same. Segments allocated by using **DosAllocSeg** must be explicitly given or retrieved by using the **DosGiveSeg** and **DosGetSeg** functions.

Parameter	Description
<i>usSize</i>	Specifies the number of bytes to be allocated. It can be any value from 0 to 65,535. If it is zero, the function allocates 65,536 bytes.
<i>pszSegName</i>	Points to a null-terminated string. The string identifies the shared memory segment and must have the following form: \sharemem\name The segment name, <i>name</i> , must have the same format as an MS OS/2 filename and must be unique. For example, the name \sharemem\public.dat is acceptable.
<i>psel</i>	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. The error value may be one of the following:

ERROR_ALREADY_EXISTS

This shared segment already exists.

ERROR_INVALID_HANDLE

The share name is invalid.

ERROR_NOT_ENOUGH_MEMORY

There is insufficient memory.

Comments

Any other process can retrieve a selector for the shared segment by using the **DosGetShrSeg** function and specifying the segment name. Although selectors for shared memory may not be the same from process to process, they do identify the same memory.

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, DosGetShrSeg

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

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

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

Parameter	Description
<i>usFrequency</i>	Specifies the repetition rate of the sound in hertz (cycles-per-second). It can be any value from 0x0025 to 0x7FFF.
<i>usDuration</i>	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. The error value may be the following:

ERROR_INVALID_FREQUENCY

The frequency is outside the range 0x0025 to 0x7FFF.

Example

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

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

■ USHORT DosBufReset(*hf*)

HFILE *hf*; file handle

The **DosBufReset** function flushes the file buffers for the specified file. It flushes the buffers by writing the current content of the file buffer to the corresponding device (for example, writing to the disk for a disk file). If the file is a disk file, the function also 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.

Parameter	Description
<i>hf</i>	Identifies the file whose buffers are to be flushed. The handle must have been previously created using the DosOpen function. If the <i>hf</i> 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. The error value may be one of the following:

ERROR_ACCESS_DENIED

A disk error occurred.

ERROR_FILE_NOT_FOUND

ERROR_INVALID_HANDLE

Comments

If the process has several open files on floppy disk, the function may have the undesirable effect of requiring the user to swap disks many times.

Example

This example opens the file *abc* and writes the contents of *achBuf* to the file. It then calls the **DosBufReset** function to flush the buffers so that the data is written to the disk before the file is closed:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesWritten;
if (!DosOpen("abc", &hf, &usAction, OL, O, 0x10, 0x41, OL)) {
    DosWrite(hf, abBuf, sizeof(abBuf), &cbBytesWritten);
    DosBufReset(hf);
}
```

See Also

DosClose, DosOpen, DosWrite

- **USHORT DosCaseMap**(*usLength, pctrycCountry, pchString*)
USHORT *usLength*; length of string to casemap
PCOUNTRYCODE *pctrycCountry*; pointer to **COUNTRYCODE** structure
PCHAR *pchString*; pointer to character string

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

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

Parameter	Description
<i>usLength</i>	Specifies the length of the given string.
<i>pctrycCountry</i>	Points to a COUNTRYCODE structure that contains the country code and the code-page identifier for the case-map operation. For a full description, see the following “Structures” section.
<i>pchString</i>	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. The error value may be one of the following:

ERROR_NLS_BAD_TYPE

An internal error occurred; the selector type does not exist.

ERROR_NLS_NO_COUNTRY_FILE

Cannot find *country.sys* file.

ERROR_NLS_NO_CTRY_CODE

The country code was not found in the *country.sys* file.

ERROR_NLS_OPEN_FAILED

Cannot open *country.sys* file.

ERROR_NLS_TABLE_TRUNCATED

The buffer is too small.

ERROR_NLS_TYPE_NOT_FOUND

An internal error occurred; the selector type is not in the file.

Structures

The **COUNTRYCODE** structure pointed to by the *pctrycCountry* parameter has the following form:

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

Field	Description
country	Specifies the country code to be used for casemapping. If zero is given, the function uses the current country code. When the function returns, the country field receives the country code that the function actually used for casemapping.

codepage Specifies the code-page identifier. It is used to determine what case-map information to use.

Comments

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

Family API 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

■ **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. This means those functions use the new directory if no explicit path is given with a filename.

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

Parameter	Description
<i>pszDirPath</i>	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 64 characters.
<i>ulReserved</i>	Specifies a reserved value. It must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 changing the directory. It does not affect the current directories of other processes.

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

Example

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

```
CHAR szDirPath[255];          /* buffer for path name */
USHORT cbDirPath = sizeof(szDirPath), usDisk;
ULONG ulLogicalDrives;
DosQCurDisk(&usDisk, &ulLogicalDrives); /* Gets current drive */
DosQCurDir(usDisk, szDirPath, &cbDirPath); /* current directory */
DosChdir("\\", OL);           /* Changes to the root directory */
.
.
DosChdir(szDirPath, OL);      /* Restores the directory */
```

See Also

DosMkdir, DosQCurDir, DosQCurDisk, DosRmdir, DosSelectDisk

- **USHORT DosChgFilePtr**(*hf, lDistance, fMethod, pulNewPtr*)
HFILE *hf*; file handle
LONG *lDistance*; distance to move
USHORT *fMethod*; method of moving (0, 1, 2)
PULONG *pulNewPtr*; new pointer location

The **DosChgFilePtr** function moves the file pointer to a new position in the file. The file pointer, maintained by the system, points to the next byte

to be read from a file or to the next position in the file to receive a written byte. **DosChgFilePtr** moves the pointer to a desired location relative to the beginning of the file, the end of the file, or the current position, as specified by the *fMethod* parameter. The distance and direction the pointer moves depends on the value and sign of the *lDistance* parameter.

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

Parameter	Description								
<i>hf</i>	Identifies the file. The handle must have been previously created by using the DosOpen function.								
<i>lDistance</i>	Specifies the number of bytes to move the file pointer in the file. If positive, the pointer moves forward through the file (beginning to end). If negative, the pointer moves from the end to the beginning.								
<i>fMethod</i>	Specifies where the move will start. It must be one of the following values:								
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>Move from the beginning of the file.</td></tr> <tr> <td>0x0001</td><td>Move from the current location.</td></tr> <tr> <td>0x0002</td><td>Move from the end of the file.</td></tr> </table>	Value	Meaning	0x0000	Move from the beginning of the file.	0x0001	Move from the current location.	0x0002	Move from the end of the file.
Value	Meaning								
0x0000	Move from the beginning of the file.								
0x0001	Move from the current location.								
0x0002	Move from the end of the file.								
<i>pulNewPtr</i>	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. The error value may be one of the following:

ERROR_INVALID_FUNCTION

The *fMethod* parameter is invalid.

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 to the end of the file, writes out the string “Hello World”, and closes the file. The *ulFilePointer* variable contains the file’s current length when the pointer is positioned at the end of the file:

```
HFILE hf;
USHORT usAction, cbBytesWritten;
ULONG ulFilePointer;
DosOpen("abc", &hf, &usAction, OL, 0, Ox11, Ox42, OL);
DosChgFilePtr(hf,                                /* file handle */
               OL,                                /* distance to move */
               2,                                  /* type of movement */
               &ulFilePointer);                   /* address of new position */
DosWrite(hf, "Hello World\n\r", 13, &cbBytesWritten);
DosClose(hf);
```

See Also

DosNewSize, DosOpen, DosRead, DosWrite

■ 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 intend to 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

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

The **DosClose** function closes the file identified by the *hf* parameter. The **DosClose** function directs the system to write the contents of all internal buffers for the file to the medium (for example, to the disk), and to update all directory information.

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

Parameter	Description
<i>hf</i>	Identifies the file to be closed. The handle must have been previously created by using the DosOpen function.

Return Value

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

ERROR_ACCESS_DENIED
 A disk error occurred.

ERROR_FILE_NOT_FOUND

ERROR_INVALID_HANDLE

Example

This example opens the file *abc*, reads in 512 bytes, and calls the **DosClose** function to close the file:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesRead;
DosOpen("abc", &hf, &usAction, OL, O, 0x01, 0x42, OL);
DosRead(hf, abBuf, sizeof(abBuf), &cbBytesRead);
DosClose(hf); /* Closes the file */
```

See Also

DosBufReset, DosFindClose, DosOpen

- **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 it, the queue content remains unchanged and the queue remains available to other processes that may have it open.

Parameter	Description
<i>hqueue</i>	Identifies the queue to be closed. It 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. The error value may be the following:

ERROR_QUE_INVALID_HANDLE
The queue handle is invalid.

Comments

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

Example

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

```
HQUEUE hqueue;  
DosCreateQueue(&hqueue, 0, "\\QUEUES\\ABC.QUE");  
.  
.  
.  
DosCloseQueue(hqueue);
```

See Also

DosCreateQueue, **DosOpenQueue**, **DosReadQueue**,
DosWriteQueue

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

The **DosCloseSem** function closes the specified system semaphore. If other processes have the semaphores open, they remain open and can be used by those processes; the semaphores cannot be used by the process that closes them. The function deletes the semaphore only when the last process using the semaphore closes it.

Parameter	Description
<i>hsem</i>	Identifies the semaphore to be closed. The 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. The error value may be one of the following:

ERROR_INVALID_HANDLE
 ERROR_SEM_IS_SET

Comments

If a process does not close its semaphores before it terminates, the system closes them. If a process owns any semaphores that the system closes for another process and the owning process is in the middle of a call to the **DosSemWait** or **DosMuxSemWait** function, the **DosSemWait** or **DosMuxSemWait** function returns immediately with the "Semaphore owner ended" error code.

Example

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

```
CHAR szSemName[] = "\\SEM\\ABC.BAR"; /* name of semaphore */
.
.
.
HSEM hsem; /* handle to semaphore */
DosOpenSem(&hsem, szSemName); /* Opens the semaphore */
.
.
.
DosCloseSem(hsem); /* Closes the semaphore */
```

See Also

DosCreateSem, DosOpenSem

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

The **DosCreateCSAlias** function creates an aliased code-segment selector for the memory segment identified by the data-segment selector, the *selDataSegment* parameter. The aliased code-segment selector may be used to pass execution control to machine instructions in the data segment.

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

Parameter	Description
<i>selDataSegment</i>	Specifies the data-segment selector.
<i>pselCodeSegment</i>	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. The error value 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 three or privilege level two (input/output privilege).

The **DosFreeSeg** function frees the aliased code-segment selector. Note that 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.

Family API Restrictions

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

- The selector returned is the actual segment address of the allocated memory.
- Freeing either the returned selector or the original selector immediately frees the block of memory.

See Also

DosAllocSeg, DosFreeSeg

- **USHORT DosCreateQueue**(*phqueue*, *fQueueOrder*, *pszQueueName*)
PHQUEUE *phqueue*; pointer to variable for queue handle
USHORT *fQueueOrder*; order elements are removed
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.

Parameter	Description								
<i>phqueue</i>	Points to the variable that receives the queue handle.								
<i>fQueueOrder</i>	Specifies the queue priority; that is, the order in which elements are to be read from and written to the queue. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>First-in/first-out (FIFO) queue. The first element put in the queue is the first element to be removed.</td></tr><tr><td>0x0001</td><td>Last-in/first-out (LIFO) queue. The last element put in the queue is the first element to be removed.</td></tr><tr><td>0x0002</td><td>Priority queue. The process that places the element in the queue specifies a priority. Elements with the highest priority are removed first.</td></tr></table>	Value	Meaning	0x0000	First-in/first-out (FIFO) queue. The first element put in the queue is the first element to be removed.	0x0001	Last-in/first-out (LIFO) queue. The last element put in the queue is the first element to be removed.	0x0002	Priority queue. The process that places the element in the queue specifies a priority. Elements with the highest priority are removed first.
Value	Meaning								
0x0000	First-in/first-out (FIFO) queue. The first element put in the queue is the first element to be removed.								
0x0001	Last-in/first-out (LIFO) queue. The last element put in the queue is the first element to be removed.								
0x0002	Priority queue. The process that places the element in the queue specifies a priority. Elements with the highest priority are removed first.								
<i>pszQueueName</i>	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. For example, the name `\queues\public.que` is acceptable.

Return Value

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

ERROR_QUE_DUPLICATE

A queue with this name already exists.

ERROR_QUE_INVALID_NAME

The queue name is invalid.

ERROR_QUE_INVALID_PRIORITY

The queue priority is invalid.

ERROR_QUE_NO_MEMORY

The queue segment is full.

Comments

The process that creates a queue owns that queue. The owner can write elements to and read elements from the queue at any time, since **DosCreateQueue** automatically opens the queue for the owner. 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 code.

See Also

DosCloseQueue, **DosOpenQueue**

- **USHORT DosCreateSem**(*fNoExclusive*, *phssm*, *pszSemName*)
USHORT *fNoExclusive*; exclusive/non-exclusive ownership flag
PHYSSEM *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 the variable pointed to by the *phssm* parameter. A system semaphore is a simple way for one process to indicate to another process a change in the status of a shared resource.

Parameter	Description
<i>fNoExclusive</i>	Specifies ownership of the semaphore. If this parameter is CSEM_PRIVATE, the process receives exclusive ownership. If it is CSEM_PUBLIC, the process does not receive exclusive ownership.
<i>phssm</i>	Points to the variable that receives the semaphore handle.
<i>pszSemName</i>	Points to a null-terminated string. The string identifies the semaphore and must have the following form: \sem\name The string name, <i>name</i> , must have the same format as an MS OS/2 filename and must be unique. For example, the name \sem\print.lck is acceptable.

Return Value

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

ERROR_ALREADY_EXISTS

A semaphore with this name already exists.

ERROR_INVALID_NAME

The semaphore name is invalid.

ERROR_INVALID_PARAMETER

Invalid NoExclusive indicator.

ERROR_TOO_MANY_SEMAPHORES

Comments

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

A process 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, and then calls **DosSemSet** to set it and **DosSemClear** to clear it:

```
HSYSEM 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**

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

The **DosCreateThread** function creates a new thread. A thread is a unique instance of execution within a given process. A thread shares the code and data segments of a process with other threads in the process, but has its own unique register values and current instruction address. The system grants CPU time to each thread, so that all threads of a process execute simultaneously.

Parameter	Description
<i>pfnFunction</i>	Points to a program-supplied function and represents the starting address of the thread. For a full description, see the following "Comments" section.
<i>ptidThread</i>	Points to the variable that receives the thread identifier.
<i>pbThrdStack</i>	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. The error value may be one of the following:

ERROR_NO_PROC_SLOTS

ERROR_NOT_ENOUGH_MEMORY

Comments

When a thread is created, the system calls the program-supplied function whose address is specified by the *pfmFunction* parameter. The system calls this function as if it were a function declared with the **far** attribute, but the system passes no argument. The program-supplied function may include local variables and may call other functions, as long as the thread's stack, specified by the *pbThrdStack* parameter has sufficient space. The stack can be allocated by using the **DosAllocSeg** function. Note that by default stacks "grow down" in memory, so the address specified by the *pbThrdStack* parameter should point to the last word in the stack, not the first. The thread terminates when the function returns or calls the **DosExit** function. The thread's stack can be freed after the thread terminates.

The program-supplied function pointed to by the *pfmFunction* parameter should have the following form:

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

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

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.

All threads within a process are identical, with the exception of the first thread created in the process. The first thread is the special signal-handler thread. If this thread terminates, signals are not processed.

The **DosCreateThread** function can create up to 255 threads per process.

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.

Example

This example sets aside a 2K buffer to be used as stack space for whatever threads are created. The first stack is set 512 bytes into the buffer (recommended minimum) and will use the data from stack area[0] to stack

DosCreateThread

area[511]. A thread is created by calling the **DosCreateThread** function. When the main execution is completed, the thread calls the **DosExit** function to terminate all threads and exit. A thread can terminate itself by calling the **DosExit** function, specifying whether to end the thread or the entire process:

```
VOID FAR Thread1();
BYTE abStackArea[2048];
.
.
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

- **USHORT DosCWait**(*fScope*, *fNoWait*, *prescResults*, *ppidProcess*,
pidWaitProcess)
- | | |
|---|--|
| USHORT <i>fScope</i> ; | flag scope |
| USHORT <i>fNoWait</i> ; | wait/no-wait flag |
| PRESULTCODES <i>prescResults</i> ; | pointer to RESULTCODES structure |
| PPID <i>ppidProcess</i> ; | pointer to variable for process identifier |
| PID <i>pidWaitProcess</i> ; | process identifier of process to wait for |

The **DosCWait** function directs the current thread to wait until the specified process has terminated. The function then 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.

Parameter	Description
<i>fScope</i>	Specifies how many processes to wait for. If it is <code>DCWA_PROCESS</code> , the thread waits until only the specified process ends. If it is <code>DCWA_PROCESSTREE</code> , the thread waits until the specified process and all its child processes end.
<i>fNoWait</i>	Specifies whether or not to wait for child processes. If it is <code>DCWW_WAIT</code> , the thread waits when child processes are still running. If it is <code>DCWW_NOWAIT</code> , the thread does not wait. This option is used to retrieve the result codes of a child process that has already ended.
<i>prescResults</i>	Points to a RESULTCODES structure that receives the termination code and result code for the child process's termination. For a full description, see the following "Structures" section.
<i>ppidProcess</i>	Points to the variable that receives the process identifier of the ending process.
<i>pidWaitProcess</i>	Specifies which process to wait for. If it is any value other than <code>0x0000</code> , the thread waits for the process with the same process identifier to end. If it is <code>0x0000</code> , 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. The error value may be one of the following:

ERROR_CHILD_NOT_COMPLETE

The child process has not terminated and the no-wait option was specified.

ERROR_INVALID_PROCID

The specified process identifier was not found.

ERROR_WAIT_NO_CHILDREN

Structures

The **RESULTCODES** structure pointed to by the *prescResults* parameter has the following form:

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

Field	Description										
codeTerminate	Specifies the system-supplied termination code and describes why the child process was terminated. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>TC_EXIT</td><td>Normal exit</td></tr><tr><td>TC_HARDERROR</td><td>Hard-error abort</td></tr><tr><td>TC_TRAP</td><td>Trap operation</td></tr><tr><td>TC_KILLPROCESS</td><td>Unintercepted DosKillProcess</td></tr></table>	Value	Meaning	TC_EXIT	Normal exit	TC_HARDERROR	Hard-error abort	TC_TRAP	Trap operation	TC_KILLPROCESS	Unintercepted DosKillProcess
Value	Meaning										
TC_EXIT	Normal exit										
TC_HARDERROR	Hard-error abort										
TC_TRAP	Trap operation										
TC_KILLPROCESS	Unintercepted DosKillProcess										
codeResult	Specifies the result code of the terminating process in its last call to the DosExit function.										

Comments

The **DosCWait** function waits for a child process to end. If the child process starts other processes, the function may also wait for these processes to end before it returns, but it will not report their status. If the specified child process has multiple threads, the function copies the result code from the last thread to the **codeResult** field of the **RESULTCODES** structure.

The **DosCWait** function returns an error if no child processes have been started. If a child process has been started but has already ended, the function may return immediately with the status, depending on the value of the *fNoWait* parameter. If no child processes have ended, the **DosCWait** function may wait until one ends before returning to the parent process.

To verify that a given return code is from a specific child process, the process identifier must be checked. Do not call the **DosCWait** function before starting the child process. If you do, the process will wait forever since a child process cannot start asynchronously.

Example

This example runs the *cmd.exe* program as a child process. After executing additional code, the example calls the **DosCWait** function to wait until *cmd.exe* terminates:

```

CHAR achFailName[128];
RESULTCODES rescResults;
PID pidProcess;
DosExecPgm(achFailName, sizeof(achFailName),
    1, "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

■ **USHORT DosDelete**(*pszFileName*, *ulReserved*)
PSZ *pszFileName*; pathname
ULONG *ulReserved*; Must be zero

The **DosDelete** function deletes a file.

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

Parameter	Description
<i>pszFileName</i>	Points to a null-terminated string that specifies the file to be deleted. The string must be a valid MS OS/2 filename.
<i>ulReserved</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_ACCESS_DENIED
 Access is denied. The directory cannot be deleted, or this is a read-only file.

ERROR_FILE_NOT_FOUND

ERROR_NOT_DOS_DISK

ERROR_PATH_NOT_FOUND

ERROR_SHARING_BUFFER_EXCEEDED

ERROR_SHARING_VIOLATION
 Another process has the file open.

Comments

Wildcard characters are not allowed in any part of the *pszFileName* parameter string. Although read-only files cannot be deleted by **DosDelete**, the **DosSetFileMode** function can be used to change read-only attributes, allowing read-only files to be deleted.

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.bar*, and displays a message reporting success or failure:

```
if (DosDelete("abc.bar", OL))
    VioWrtTty("abc.bar not deleted\n\r", 21, 0);
else
    VioWrtTty("abc.bar deleted\n\r", 17, 0);
```

See Also

DosRmdir, **DosSetFileMode**

- **USHORT DosDevConfig**(*pvoidDevInfo*, *usItem*, *usReserved*)
PVOID *pvoidDevInfo*; 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.

Parameter	Description								
<i>pvoidDevInfo</i>	Points to the variable that receives device information. The type of information depends on the <i>usItem</i> parameter.								
<i>usItem</i>	Specifies what device information to retrieve. It can be one of the following values:								
	<table><tr><th>Value</th><th>Device information</th></tr><tr><td>0x0000</td><td>Number of printers attached.</td></tr><tr><td>0x0001</td><td>Number of RS232 cards attached.</td></tr><tr><td>0x0002</td><td>Number of floppy-disk drives installed.</td></tr></table>	Value	Device information	0x0000	Number of printers attached.	0x0001	Number of RS232 cards attached.	0x0002	Number of floppy-disk drives installed.
Value	Device information								
0x0000	Number of printers attached.								
0x0001	Number of RS232 cards attached.								
0x0002	Number of floppy-disk drives installed.								

0x0003	Math coprocessor: FALSE if coprocessor not present; TRUE if math coprocessor is present.
0x0004	PC submodel type: system submodel byte.
0x0005	PC model type: system model byte.
0x0006	Primary display adapter type: FALSE if monochrome/printer adapter; TRUE for other display adapter.

usReserved Specifies a reserved value. It must be zero.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 information */
             3,          /* item number */
             0);         /* reserved */
if (bDevInfo)
    VioWrtTty("Math coprocessor present\n\r", 26, 0);
else
    VioWrtTty("Math coprocessor not present\n\r", 30, 0);

```

See Also

DosDevIOCtl, **VioGetConfig**

■ **USHORT DosDevIOCtl**(*pvoidData*, *pvoidParms*, *usFunction*, *usCategory*, *hDevice*)

PVOID <i>pvoidData</i> ;	data area
PVOID <i>pvoidParms</i> ;	command arguments
USHORT <i>usFunction</i> ;	device function
USHORT <i>usCategory</i> ;	device category
USHORT <i>hDevice</i> ;	handle of device

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

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

Parameter	Description
<i>pvoidData</i>	Points to a buffer that contains any data required for the given control function. The DosDevIOCtl function may copy data to the buffer.
<i>pvoidParms</i>	Points to a buffer that contains any data required for the given control function. The DosDevIOCtl function may copy data to the buffer.
<i>usFunction</i>	Specifies the device-control function. It can be any one of the device-control function codes described in Chapter 4, "Input-and-Output Control Functions."
<i>usCategory</i>	Specifies the device categories. It can be any one of the device categories described in Chapter 4, "Input-and-Output Control Functions."
<i>hDevice</i>	Identifies the device that receives the device-control function. The handle must have been previously created by using the DosOpen function or 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 to 0xFFFF are user-dependent error values. Values in the range 0xFE00 to 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

Family API 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 typeamatic rate of the keyboard. Before you can use the **DosDevIOCtl** function to access the keyboard, you must first 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(OL,                 /* data area */
            (PCHAR) usParameters, /* command arguments */
            0x54,                /* function code */
            4,                   /* device category */
            hkbd);               /* handle to device keyboard */
```

See Also

DosOpen

■ **USHORT DosDupHandle(*hfOld*, *phfNew*)**
HFILE *hfOld*; existing file handle
PHFILE *phfNew*; new file handle

The **DosDupHandle** function duplicates an existing file handle. The new handle has all the handle-specific information as the existing handle, such as file-pointer position and access method. The original handle and the duplicate are interchangeable since changes to one affect the other. For example, moving the file pointer for the original handle moves the pointer for the new handle.

The variable pointed to by the *phfNew* parameter specifies the value of the new handle. The **DosDupHandle** function accepts a handle value and uses that value for the new handle. The value must already be an open file handle. The **DosDupHandle** function closes that file before duplicating

DosDupHandle

the handle. If the *phfNew* parameter is 0xFFFF, **DosDupHandle** chooses its own file handle value and copies it to the variable pointed to by the *phfNew* parameter.

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

Parameter	Description
<i>hfOld</i>	Identifies the file whose handle is to be duplicated. The handle must have been previously created by using the DosOpen function.
<i>phfNew</i>	Points to the variable that contains the desired file handle. If this parameter is 0xFFFF, the function creates a new handle and copies it to the variable pointed to by the parameter. For any other value, that value itself is used as the handle.

Return Value

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

ERROR_INVALID_HANDLE

ERROR_INVALID_TARGET_HANDLE

ERROR_TOO_MANY_OPEN_FILES

Comments

Closing the original handle by using the **DosClose** function does not close the duplicate handle; closing the duplicate does not close the original. The corresponding file is not closed until the last handle is closed.

The **DosSetFHandleState** function changes the handle flags for the duplicate function. This is useful for changing inheritance flags or access modes of duplicate handles for child processes.

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; /* new handle without closing old handle */
DosDupHandle(1, &hfNew); /* duplicate standard out */
DosWrite(hfNew, "Hello World\n\r", 13, &cbBytesWritten);
```

See Also

DosChgFilePtr, **DosClose**, **DosExecPgm**, **DosMakePipe**, **DosRead**,
DosWrite

■ **VOID DosEnterCritSec(VOID)**

The **DosEnterCritSec** function temporarily prevents other threads in the current process from executing. Other threads in the same process will not execute until the current thread issues the **DosExitCritSec** function.

This function has no parameters.

Return Value

This function does not return a value.

Comments

If a signal occurs, thread 1 may begin executing to process the signal while another thread in the same process calls the **DosEnterCritSec** function. Any processing that thread 1 does to satisfy the signal must not access the critical resource protected by the **DosEnterCritSec** function.

For information on thread 1, see the **DosCreateThread** function.

See Also

DosCreateThread, **DosExitCritSec**, **DosSetSigHandler**

■ **USHORT DosErrClass(usErrorCode, pusClass, pfsAction, pusLocus)**

USHORT <i>usErrorCode</i> ;	error code for analysis
PUSHORT <i>pusClass</i> ;	pointer to variable for error classification
PUSHORT <i>pfsAction</i> ;	pointer to variable for action
PUSHORT <i>pusLocus</i> ;	pointer to variable for error origin

The **DosErrClass** function helps MS OS/2 programs recognize and process error codes that they receive from MS OS/2 functions. The **DosErrClass** function retrieves a classification of the error and a recommended action. Depending on the program, you can follow the recommended action or try a more specific recovery method.

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

Parameter	Description
<i>usErrorCode</i>	Specifies the error code returned by an MS OS/2 function.
<i>pusClass</i>	Points to the variable that receives the classification of that error. It can be one of the following values:
Value	Meaning
ERRCLASS_ALREADY	
ERRCLASS_APPERR	A probable application error has occurred.
ERRCLASS_AUTH	Authorization has failed.
ERRCLASS_BADFMT	
ERRCLASS_CANT	Cannot perform requested action.
ERRCLASS_HRDFAIL	A device hardware failure has occurred.
ERRCLASS_INTRN	An internal error has occurred.
ERRCLASS_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. It 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. It 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 this function returns an error, the **DosErrClass** function is called to determine the class of the error. If the error is a device hardware failure, the program exits. A device hardware error could be caused by a disk-drive door being open or a non-existent disk drive, for example.

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

See Also

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, the disk-drive door being opened while a floppy disk is being read results in a hard error.

The **DosError** function disables the default processing by foregoing the message and directing any function that encounters a hard error or exception to return an appropriate error code. At that point, the process must determine the appropriate action.

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

Parameter	Description
<i>fEnable</i>	Specifies whether to disable or enable processing. It can be one of the following values:

Value	Meaning
0x0000	Disable hard-error processing.
0x0001	Enable hard-error processing.
0x0002	Disable exception processing; 0x0000 or 0x0001 enables exception processing.

Return Value

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

ERROR_INVALID_DATA

The action flag is invalid.

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.

Family API Restrictions

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

- If the *fEnable* parameter is 0x0000, all subsequent **int 24h** requests fail until a subsequent call is made to the **DosError** function with *fEnable* set to 0x0001.

Example

This example calls the **DosError** function to turn off processing of hard errors. It then calls the **DosErrClass** function to process any error that is received:

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

See Also

DosErrClass, **DosSetFHandState**

■ **USHORT DosExecPgm**(*pchFailName*, *cbFailName*, *fExecFlags*,
pszArgs, *pszEnv*, *prescResults*, *pszPgmName*)

PCHAR *pchFailName*; pointer to buffer for failed filename
USHORT *cbFailName*; size of failed filename buffer
USHORT *fExecFlags*; synchronous/trace flags
PSZ *pszArgs*; pointer to argument strings
PSZ *pszEnv*; pointer to environment strings
RESULTCODES *prescResults*; pointer to structure for result codes
PSZ *pszPgmName*; pointer to program name to execute

The **DosExecPgm** function loads and starts another program. The new program is called a child process, and the starting process is called the parent process.

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

Parameter	Description
<i>pchFailName</i>	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.
<i>cbFailName</i>	Specifies the length in bytes of the buffer pointed to by the <i>pchFailName</i> parameter.
<i>fExecFlags</i>	Specifies how a given program should be run. It can be one of the following values:
Value	Meaning
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 <i>prescResults</i> parameter.
EXEC_ASYNC	Execute asynchronously to the parent process. The DosExecPgm function copies the process identifier of the child process to the first field of the structure pointed to by the <i>prescResults</i> parameter.
EXEC_ASYNCRESULT	Execute asynchronously to the parent process. Before returning, the DosExecPgm function copies the process identifier to the codeTerminate field of the structure

pointed to by the *prescResults* parameter. When the child process ends, the system saves its termination and result codes. The parent process can retrieve these values by using the **DosCWait** function.

EXEC_TRACE

Execute under conditions for tracing. The parent process debugs the child process.

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 keyboard input or screen output, other than through the **VioPopUp** function. It also should not call virtual I/O, keyboard, or mouse functions.

<i>pszArgs</i>	Points to a set of null-terminated argument strings. If the <i>pszArgs</i> parameter is zero, no argument strings are passed to the process.
<i>pszEnv</i>	Points to a set of null-terminated environment strings. If the <i>pszEnv</i> parameter is zero, the child process inherits the environment of the parent process unchanged.
<i>prescResults</i>	Points to the RESULTCODES structure that receives the termination and result codes of the child process. For a full description, see the following "Structures" section.
<i>pszPgmName</i>	Points to a null-terminated string that specifies the process to load and start. The string must be a valid MS OS/2 filename, including filename extension. It must specify an executable file.

Return Value

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

ERROR_ACCESS_DENIED

A directory cannot be accessed.

ERROR_AUTODATASEG_EXCEEDS_64k

ERROR_BAD_ENVIRONMENT

ERROR_BAD_FORMAT

The file is not executable.

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

An invalid *fExecFlags* parameter was specified.

ERROR_INVALID_MINALLOCSIZE

ERROR_INVALID_MODULETYPE

ERROR_INVALID_ORDINAL

ERROR_INVALID_SEGMENT_NUMBER

ERROR_INVALID_SEGDPL

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

Another process has the specified file open in a mode that prevents it from being shared.

ERROR_TOO_MANY_OPEN_FILES

Structures

The **RESULTCODES** structure pointed to by the *prescResults* parameter has the following form:

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

Field	Description										
codeTerminate	Specifies the child process identifier (PID) 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: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>TC_EXIT</td><td>Normal exit</td></tr> <tr> <td>TC_HARDERROR</td><td>Hard-error abort</td></tr> <tr> <td>TC_TRAP</td><td>Trap operation</td></tr> <tr> <td>TC_KILLPROCESS</td><td>Unintercepted DosKillProcess</td></tr> </table>	Value	Meaning	TC_EXIT	Normal exit	TC_HARDERROR	Hard-error abort	TC_TRAP	Trap operation	TC_KILLPROCESS	Unintercepted DosKillProcess
Value	Meaning										
TC_EXIT	Normal exit										
TC_HARDERROR	Hard-error abort										
TC_TRAP	Trap operation										
TC_KILLPROCESS	Unintercepted DosKillProcess										
codeResult	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.										

Comments

The *pszPgmName* parameter points to the MS OS/2 filename of the program to load and start. 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 if the filename is not found in the current directory, the **DosExecPgm** function searches each directory specified in the **PATH** environment variable in the environment of the parent process for the given file. Note that any filename extension is acceptable, although the extension *.exe* is most common.

The **DosExecPgm** function directs the child process to execute synchronously or asynchronously. If the child process is synchronous, the parent process waits while the child process executes and does not continue executing until the child process ends. If the child process is asynchronous, the parent and child processes execute simultaneously.

To help the parent process identify the child process and determine when and how the child process ended execution, the **DosExecPgm** function copies information such as the child process identifier or termination code to the structure pointed to by the *prescResults* parameter. The parent process may use the child process identifier to control the child process or request information from it. For example, the **DosCWait** function can be used to retrieve the termination code from the child process, or to make the parent process wait until the child process ends.

The **DosExecPgm** function passes an argument string to the child process. It copies the argument string pointed to by the *pszArgs* parameter to the process's environment segment. This string represents the command parameters for the program. The string can have any format but must end with two null characters (double zero). A typical format is the program name, a null character, the program parameters (separated by spaces), and two null characters.

The **DosExecPgm** function passes environment strings to the child process. It copies the environment strings pointed to by the *pszEnv* parameter to the process's environment segment. These strings represent environment variables and their current values. An environment string has the following form:

variable=value

The string ends with a null character. Two or more strings can be concatenated to pass multiple environment strings to the child process. In any case, the last string must end with two null characters (double zero).

If the *fExecFlags* parameter is **EXEC_ASYNCRESULT**, the system reserves memory to store the termination and result codes of the child process. This memory remains allocated until the parent process calls the **DosCWait** function to retrieve the information. To conserve memory, do not set *fExecFlags* to **EXEC_ASYNCRESULT**, unless the parent process retrieves the codes.

The child process receives a separate and distinct address space (that is, it receives its own local descriptor table). This means that the parent process cannot access data in the child process and vice versa. To pass data between processes, the parent process typically opens a pipe by using the **DosMakePipe** function before starting the child process and 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 from the parent process all open file handles except those opened with the inheritance flag set to 0x0000 (for more information, see the **DosOpen** function). It also inherits all pipe handles.

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 writes to the standard output, the data goes to the file, not to the screen.

Family API Restrictions

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

- The only value allowed for the *fExecFlags* parameter is EXEC_SYNC. Other values cause errors.
- 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 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 simultaneously with the main program:

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

See Also

DosCWait, **DosExit**, **DosGetEnv**, **DosKillProcess**

- **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 as specified by the *fTerminate* parameter.

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

Parameter	Description
<i>fTerminate</i>	Specifies whether to terminate the current thread or the process and/or 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.
<i>usExitCode</i>	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 thread is the last one in the process, the process also ends. If *fTerminate* is **EXIT_PROCESS**, the function terminates all threads in the process but creates one last thread temporarily to execute 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 and supplies the exit code specified by the last call to the **DosExit** function to the parent process through the **DosCWait** function.

Family API Restrictions

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

- The function exits from the currently executing program since there are no threads in the real-mode environment. In other words, if the *fTerminate* parameter is **EXIT_THREAD**, the entire process, not just a thread, ends.

Example

This example creates a thread, referred to as thread 2. Thread 1, the main process, exits by calling the **DosExit** function with the first parameter set to **EXIT_PROCESS** to stop all threads. Thread 2, the thread created with

the call to **DosCreateThread**, ends its processing by calling **DosExit** with the first parameter set to **EXIT_THREAD** so that only that thread will be terminated:

```

BYTE bStackArea[2048];
.
.
.
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

■ VOID DosExitCritSec(VOID)

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

This function has no parameters.

Return Value

This function does not return a value.

Comments

To restore normal thread execution to other threads of a process, use the **DosExitCritSec** function after the **DosEnterCritSec** function.

See Also

DosCreateThread, DosEnterCritSec

■ **USHORT DosExitList**(*fFnCode*, *pfnFunction*)
USHORT *fFnCode*; function code
VOID FAR * *pfnFunction*(**USHORT**); address of function

The **DosExitList** function lists functions that are to be executed when the current process ends. The **DosExitList** function specifies a termination function that gains control when the process completes execution. The termination function may define multiple termination functions that receive control when a process terminates. When the process terminates, MS OS/2 transfers control to each function on the list. If there are multiple functions on the list, they each get control, in indeterminate order.

Parameter	Description								
<i>fFnCode</i>	Specifies whether to add or remove a routine address from the list. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>EXLST_ADD</td><td>Add function to termination list.</td></tr><tr><td>EXLST_REMOVE</td><td>Remove function from termination list.</td></tr><tr><td>EXLST_EXIT</td><td>Termination processing complete; call the next function on termination list.</td></tr></table>	Value	Meaning	EXLST_ADD	Add function to termination list.	EXLST_REMOVE	Remove function from termination list.	EXLST_EXIT	Termination processing complete; call the next function on termination list.
Value	Meaning								
EXLST_ADD	Add function to termination list.								
EXLST_REMOVE	Remove function from termination list.								
EXLST_EXIT	Termination processing complete; call the next function on termination list.								
<i>pfnFunction</i>	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. The error value may be one of the following:

ERROR_INVALID_DATA
The action specified is out of range.
ERROR_NOT_ENOUGH_MEMORY

Comments

Library modules typically use the **DosExitList** function; it allows them to free resources or clear flags and semaphores in case the client process terminates without notifying them.

The termination function must be defined in the process (that is, it must be in a code segment that belongs to the process). The termination function has one parameter but returns no value. It should have the following form:

```
VOID FAR <FuncName>(<usTermCode>)
USHORT <usTermCode>;
{
    .
    .
    .
    DosExitList(EXLST_EXIT, 0);
}
```

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

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

The system passes control to the termination function by using a **jmp** instruction, so the function must not execute a **return** statement. Instead, it must call the **DosExitList** function to end.

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

It is critical that the termination functions be as short and fail-safe as possible. A termination function must call the **DosExitList** function to end; otherwise, the process “hangs” since MS OS/2 is not able to terminate it.

When the termination functions are executed, the process is already in a state of partial termination. To ensure good response to a user request to end a program, there should be a minimum delay in allowing termination to complete. All threads except for the one executing the **DosExitList** function have been destroyed.

In general, 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, telling it to add the locally defined function **CleanUp** to the list of routines to be called when the process terminates. The function **CleanUp** displays a message that it is cleaning up, and then calls **DosExitList**, telling that it has completed its work and can be terminated:

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

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

See Also

DosCreateThread, **DosExecPgm**, **DosExit**

- **USHORT DosFileLocks**(*hf*, *plUnlockRange*, *plLockRange*)
HFILE *hf*; file handle
PLONG *plUnlockRange*; pointer to range to be unlocked
PLONG *plLockRange*; 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 that might have that file open from accessing the locked region.

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

Parameter	Description
<i>hf</i>	Specifies the file handle. It must have been previously opened by using the DosOpen function.

plUnlockRange Points to two consecutive 32-bit integers that specify the file offset and the number of bytes of the file to be unlocked. For a full description, see the following “Structures” section.

plLockRange Points to two consecutive 32-bit integers that specify the file offset and the number of bytes of the file to be locked. For a full description, see the following “Structures” section.

Return Value

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

ERROR_INVALID_HANDLE

ERROR_LOCK_VIOLATION

Structures

The structure pointed to by the *plUnlockRange* and *plLockRange* parameters has the following form:

```
struct {
    LONG lFileOffset;
    LONG lRangeLength;
};
```

Field	Description
lFileOffset	Specifies the offset from the beginning of the file to the start of the area to lock or unlock.
lRangeLength	Specifies the length in bytes of the area to lock or unlock.

Comments

The **DosFileLocks** function can both lock and unlock regions. If an unlocking range is given, that region is unlocked first, then any specified locking region is locked. 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 the file, and attempting to lock beyond the end of the file does not result in an error.

A process should not lock a region of a file any longer than it needs to access that region. If the process ends without unlocking a file, the system closes the file and releases the locks.

Handles created by using the **DosDupHandle** function allow access to the locked regions when used in the same process. Child processes do not inherit access to locked regions.

The proper method for using locks is to attempt to lock the desired region and examine the error code. If the code indicates an error, that region is already locked.

The **DosFileLocks** function should be used only when a file is opened using the deny-read or deny-none sharing modes, or the file is opened for read/write access and deny-write sharing mode.

Example

This example opens the file *abc.ext* and calls the **DosFileLocks** function to lock 100 bytes of the file starting with byte number 3. No other file may read or write to this range within the file until **DosFileLocks** is called to unlock the range or until the file is closed:

```
struct RANGE {
    LONG lFileOffset;
    LONG lRangeLength;
};

.
.
.
struct RANGE rngRange;
HFILE hf;
USHORT usAction;
rngRange.lFileOffset = 3L; /* offset to begin lock */
rngRange.lRangeLength = 100L; /* range to lock */
DosOpen("abc.ext", &hf, &usAction, OL, O, Ox01, Ox42, OL);
DosFileLocks(hf, /* handle of file to lock */
              OL, /* unlock range (NULL to disable) */
              (PLONG) &rngRange); /* address of lock range */
.
.
.
DosFileLocks(hf, /* handle of file to unlock */
              (PLONG) &rngRange, /* address of unlock range */
              OL); /* lock range (NULL to disable) */
```

See Also

DosExecPgm, **DosOpen**



USHORT DosFindClose(hdir)

HDIR hdir;

directory-search handle

The **DosFindClose** function closes the search directory identified by the directory handle specified by the *hdir* parameter. The **DosFindFirst** and

DosFindNext functions use the directory-search handle to locate files whose names match a given name.

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

Parameter	Description
<i>hdir</i>	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. The error value may be the following:

ERROR_INVALID_HANDLE

Family API Restrictions

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

- After closing a directory handle, any attempt to use the handle in a subsequent call to the **DosFindNext** function causes that function to return an error code.
- The handle must be reopened by using the **DosFindFirst** function.

Example

This example calls the **DosFindFirst** function to find all files that match “*.*”. When processing is done with the list of files, the handle is closed by calling the **DosFindClose** function:

```
HDIR hdir = 0xffff;
USHORT usSearchCount = 1;
FILEFINDBUF findbuf;
DosFindFirst("*.*", &hdir, 0x00, &findbuf,
             sizeof(findbuf), &usSearchCount, OL);
.
.
DosFindClose(hdir);                /* Closes the directory */
```

See Also

DosFindFirst, **DosFindNext**, **DosSearchPath**

- **USHORT DosFindFirst**(*pszFileSpec*, *phdir*, *usAttribute*, *pfindbuf*,
usBufLen, *pusSearchCount*, *ulReserved*)
 - PSZ** *pszFileSpec*; file specification
 - PHDIR** *phdir*; pointer to variable for handle
 - USHORT** *usAttribute*; search attribute
 - PFILEFINDBUF** *pfindbuf*; pointer to result buffer
 - USHORT** *usBufLen*; length of result buffer
 - PUSHORT** *pusSearchCount*; files to find/files found
 - ULONG** *ulReserved*; Must be zero

The **DosFindFirst** function searches a directory for the first file or first part of a group of files that have names and attributes matching the specified name and attributes. The function copies the names and directory information of the file or files to the **Filefindbuf** structure pointed to by the *pfindbuf* parameter. The information returned is guaranteed to be at least as accurate as the most recent call to the **DosClose** or **DosBufReset** function.

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

Parameter	Description
<i>pszFileSpec</i>	Points to a null-terminated string. The string must be a valid MS OS/2 pathname and may contain wildcard characters.
<i>phdir</i>	Points to the variable that contains the directory-search request. If this parameter is 0x0001, the system default directory-search handle is used. If it is 0xFFFF, the search directory that is used by the process is allocated, and the function copies a new directory-search handle to the unsigned variable. The handle can be used in subsequent calls to the DosFindNext function.
<i>usAttribute</i>	Specifies the file attribute of the file to be located. It can be a combination of the following values:
Value	Meaning
0x0000	Search for normal files.
0x0001	Search for read-only files.
0x0002	Search for hidden files.
0x0004	Search for system files.
0x0010	Search for subdirectories.
0x0020	Search for archived files.

<i>pfindbuf</i>	Points to a FILEFINDBUF structure that receives the result of the search. For a full description, see the following “Structures” section.
<i>usBufLen</i>	Specifies the length in bytes of the structure pointed to by the <i>pfindbuf</i> parameter.
<i>pusSearchCount</i>	Points to a variable that specifies the number of matching filenames to locate. The function copies the actual number of filenames found to the unsigned variable before returning.
<i>ulReserved</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_BUFFER_OVERFLOW

ERROR_DRIVE_LOCKED

ERROR_FILE_NOT_FOUND

A bad character was found in the final part of the path.

ERROR_INVALID_HANDLE

ERROR_INVALID_PARAMETER

The search attribute is invalid.

ERROR_NO_MORE_FILES

ERROR_NO_MORE_SEARCH_HANDLES

ERROR_NOT_DOS_DISK

ERROR_PATH_NOT_FOUND

Structures

The **FILEFINDBUF** structure pointed to by the *pfindbuf* parameter 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;
```

Field	Description
fdateCreation	Specifies the date of file creation.
ftimeCreation	Specifies the time of file creation.
fdateLastAccess	Specifies the date of the last file access.
ftimeLastAccess	Specifies the time of the last file access.
fdateLastWrite	Specifies the date of the last write to the file.
ftimeLastWrite	Specifies the time of the last write to the file.
cbFile	Specifies the end of file data.
cbFileAlloc	Specifies the file size allocated.
attrFile	Specifies the file attribute.
cchName	Specifies the length of the null-terminated filename.
achName[13]	Specifies the null-terminated filename.

Comments

The *pusSearchCount* parameter specifies the number of files to search for. The actual 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 function fails to find a match or cannot copy all information about the file to the structure, it returns an error.

The **DosFindFirst** function creates a directory-search handle. This handle can be used in subsequent calls to the **DosFindNext** function to specify the directory and filename to be searched. Each call to the **DosFindFirst** function automatically closes the specified directory-search handle if it has not been previously closed 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. Do not depend on the exact length of the filename. This may change in future versions of MS OS/2.

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

Family API Restrictions

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

- The *phdir* parameter must be set to 0x0001.
- Subsequent calls to **DosFindFirst** must have a *phdir* parameter of 0x0001 unless the **DosFindClose** function has been called. In that case, 0x0001 or 0xffff are allowed.

Example

This example calls the **DosFindFirst** function to find the file *abc.ext*. It displays an error if the file is not found:

```

HDIR hdir = 0xffff;
USHORT usSearchCount = 1;
FILEFINDBUF findbuf;
if (DosFindFirst("abc.ext",          /* filename to search for */
                &hdir,              /* address of directory handle */
                0x00,                /* 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\n\r", 16, 0);
else {

```

See Also

DosBufReset, **DosClose**, **DosFindClose**, **DosFindNext**,
DosQFileMode, **DosQFSInfo**

- **USHORT DosFindNext**(*hdir*, *pfindbuf*, *cbFindBuf*, *pusSearchCount*)
HDIR *hdir*; handle of directory search
PFILEFINDBUF *pfindbuf*; pointer to result buffer
USHORT *cbFindBuf*; length of result buffer
PUSHORT *pusSearchCount*; files to find/files found

The **DosFindNext** function searches for the next file or group of files matching the filename and attributes specified by the directory-search handle specified by the *hdir* parameter. The function copies the names and directory information of the file or files to the **FILEFINDBUF** structure pointed to by the *pfindbuf* parameter. The information returned is guaranteed to be at least as accurate as the most recent call to the **DosClose** or **DosBufReset** function.

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

Parameter	Description
<i>hdir</i>	Identifies the search directory. It must have been previously created by using the DosFindFirst function.
<i>pfindbuf</i>	Points to a FILEFINDBUF structure that receives the result of the search. For a full description, see the following “Structures” section.
<i>cbFindBuf</i>	Specifies the length in bytes of the structure pointed to by the <i>pfindbuf</i> parameter.
<i>pusSearchCount</i>	Points to an unsigned variable that specifies the number of matching filenames to locate. The function copies the actual 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. The error value may be one of the following:

ERROR_BUFFER_OVERFLOW

ERROR_INVALID_HANDLE

ERROR_INVALID_PARAMETER

A search count of zero was specified.

ERROR_NO_MORE_FILES

ERROR_NOT_DOS_DISK

Structures

The **FILEFINDBUF** structure pointed to by the *pfindbuf* parameter 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;
```

Field	Description
fdateCreation	Specifies the date of file creation.
ftimeCreation	Specifies the time of file creation.
fdateLastAccess	Specifies the date of last file access.
ftimeLastAccess	Specifies the time of last file access.
fdateLastWrite	Specifies the date of the last write to the file.
ftimeLastWrite	Specifies the time of the last write to the file.
cbFile	Specifies the end of file data.
cbFileAlloc	Specifies the file size allocated.
attrFile	Specifies the file attribute.
cchName	Specifies the length of the null-terminated filename.
achName[13]	Specifies the null-terminated filename.

Comments

The *pusSearchCount* parameter specifies the number of files to search for. The actual 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 function fails to find a match or cannot copy all 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. Do not depend on the exact length of the filename. It may change in future versions of MS OS/2.

Family API Restrictions

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

- The *hdir* parameter must always be 0x0001.

Example

This example calls the **DosFindFirst** function to find all files matching “*.*”, and then displays them one at a time using the **DosFindNext** function:

```
HDIR hdir = 0xFFFF;
USHORT usSearchCount = 1;
FILEFINDBUF findbuf;
DosFindFirst("*.*", &hdir, 0x00, &findbuf,
    sizeof(findbuf), &usSearchCount, OL);
do {
    VioWrtTTY(findbuf.achName, findbuf.cchName, 0);
    VioWrtTTY("\n\r", 2, 0);    /* cursor to next line */
}
while (!DosFindNext(hdir,    /* handle to directory */
    &findbuf,                /* address of buffer */
    sizeof(findbuf),        /* length of buffer */
    &usSearchCount));        /* number of files to find */
```

See Also

DosFindClose, DosFindFirst

- **USHORT DosFlagProcess**(*pidProcess*, *fScope*, *usFlagNum*, *usFlagArg*)
PID *pidProcess*; identifier of process to 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 process. By default, the process ignores these signals, but if it expects the event flag signal, it can respond to these signals by defining a signal handler by using the **DosSetSigHandler** function. A process can also refuse event flag signals, causing the **DosFlagProcess** function to return an error code.

Parameter	Description
<i>pidProcess</i>	Specifies the process identifier of the process that receives the flag.
<i>fScope</i>	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.
<i>usFlagNum</i>	Specifies the number of the flag to set. It can be one of the following values:

	Value	Meaning
	PFLG_A	Process flag A.
	PFLG_B	Process flag B.
	PFLG_C	Process flag C.
<i>usFlagArg</i>		Specifies an argument to be passed to the specified process.

Return Value

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

ERROR_INVALID_FLAG_NUMBER

ERROR_INVALID_FUNCTION

The action specified is out of range.

ERROR_INVALID_PROCID

ERROR_SIGNAL_REFUSED

The target process has refused this signal.

Comments

The error message that tells you the signal is already pending indicates that a signal of the type just sent is already waiting to be processed; therefore, the current signal cannot be accepted.

Example

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

```
CHAR achFailName[128];
RESULTCODES rescResults;
DosExecPgm(achFailName, sizeof(achFailName),
    1, "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**

- **USHORT DosFreeModule(*hmod*)**
HMODULE *hmod*; module handle

The **DosFreeModule** function frees the specified dynamic-link module, making the given *hmod* parameter no longer valid. After a process has freed a module, any function addresses it may have retrieved from the module are no longer valid and cause a protection fault if they are called.

Parameter	Description
<i>hmod</i>	Identifies the dynamic-link module to be freed. It must have been previously created by using the DosLoadModule function.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 actually remove a module from memory until it is no longer used by any process.

See Also

DosLoadModule

- **USHORT DosFreeSeg(*sel*)**
SEL *sel*; selector

The **DosFreeSeg** function frees the memory segment specified by the *sel* parameter. The function accepts selectors for memory segments, shared-memory segments, huge-memory segments, and aliased code segments. For shared-memory segments, **DosFreeSeg** does not actually free the segment until the last process accessing the segment frees it. For aliased code segments, **DosFreeSeg** frees the code-segment selector, but the corresponding data-segment selector remains valid until it is freed.

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

Parameter	Description
<i>sel</i>	Specifies the selector of the segment to be freed.

Return Value

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

ERROR_ACCESS_DENIED

An invalid or global descriptor table (GDT) selector was specified.

Family API 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 not unique. Freeing one frees both.

Example

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

```
SEL sel;
DosAllocHuge(3, 200, &sel, 0, 5);
.
.
.
DosFreeSeg(sel);
```

See Also

DosAllocHuge, **DosAllocSeg**, **DosAllocShrSeg**, **DosCreateCSAlias**

- **USHORT** **DosGetCollate**(*cbBuf*, *pctrycCountry*, *pchBuf*, *pcbTable*)
USHORT *cbBuf*; size of buffer
PCOUNTRYCODE *pctrycCountry*; pointer to **COUNTRYCODE** structure
PCHAR *pchBuf*; pointer to buffer for table
PUSHORT *pcbTable*; length of returned table

The **DosGetCollate** function retrieves the collating sequence table for the given country code and code-page identifier. The collating sequence table

is a 256-element character array 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 do not have to be 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 function **DosGetCollate** copies the collating sequence table from the *country.sys* file to the buffer pointed to by the *pchBuf* parameter. If the buffer is too small to hold all the information, **DosGetCollate** truncates the information. If the buffer is larger than the information, the function fills any remaining bytes with zeros.

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

Parameter	Description
<i>cbBuf</i>	Specifies the size in bytes of the buffer that receives the collating sequence table.
<i>pctrycCountry</i>	Points to a COUNTRYCODE structure that contains the country code and the code-page identifier to be used to retrieve the collating sequence table. For a full description, see the following “Structures” section.
<i>pchBuf</i>	Points to the buffer that receives the collating sequence table.
<i>pcbTable</i>	Points to the variable that receives the actual 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.

Structures

The **COUNTRYCODE** structure pointed to by the *pctrycCountry* parameter has the following form:

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

Field	Description
country	Specifies the country code to be used to retrieve the collating sequence table. If zero is given, the function uses the current country code. When the function returns, the country field receives the country code that the function actually used.
codepage	Specifies the code-page identifier to be used to retrieve the collating sequence table.

Comments

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

Family API 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**

■ **USHORT** **DosGetCp**(*cbBuf*, *pusBuf*, *pcbCodePgLst*)
USHORT *cbBuf*; length of buffer for list
PUSHORT *pusBuf*; pointer to buffer for list
PUSHORT *pcbCodePgLst*; pointer to variable for length of returned list

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, it inherits its current code page from its parent process.

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

Parameter	Description
<i>cbBuf</i>	Specifies the length in bytes of the code-page list.
<i>pusBuf</i>	Points to the buffer that receives the code-page list.
<i>pcbCodePgLst</i>	Points to the variable that receives the actual 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.

See Also

DosSetCp

- **USHORT** **DosGetCtryInfo**(*cbBuf*, *pctrycCountry*, *pctryiBuf*, *pcbCountryInfo*)
USHORT *cbBuf*; length of data area
PCOUNTRYCODE *pctrycCountry*; address of **COUNTRYCODE** structure
PCOUNTRYINFO *pctryiBuf*; pointer to data area
PUSHORT *pcbCountryInfo*; pointer to length of returned data

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

The function copies the information from the *pcountry.sys* file to the **COUNTRYINFO** structure pointed to by the *pctryiBuf* parameter. If the structure is too small to hold all the information, **DosGetCtryInfo** truncates the information. If the structure is larger than the information requires, the function fills any remaining bytes with zeros.

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

Parameter	Description
<i>cbBuf</i>	Specifies the size in bytes of the COUNTRYINFO structure pointed to by the <i>pctryiBuf</i> parameter.
<i>pctrycCountry</i>	Points to the COUNTRYCODE structure that contains the country code and the code-page identifier to be used to retrieve country-dependent information. For a full description, see the following "Structures" section.

pctryiBuf Points to the **COUNTRYINFO** structure that receives the country-dependent formatting information. For a full description, see the following “Structures” section.

pcbCountryInfo Points to the variable that receives the actual 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. The error value may be one of the following:

ERROR_NLS_BAD_TYPE
An internal error occurred; the selector type does not exist.

ERROR_NLS_NO_COUNTRY_FILE
Cannot find the *country.sys* file.

ERROR_NLS_NO_CTRY_CODE
The country code was not found in the *country.sys* file.

ERROR_NLS_OPEN_FAILED
Cannot open the *country.sys* file.

ERROR_NLS_TABLE_TRUNCATED
The buffer is too small.

ERROR_NLS_TYPE_NOT_FOUND
An internal error occurred; selector type is not in the file.

Structures

The **COUNTRYCODE** structure pointed to by the *pctrycCountry* parameter has the following form:

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

Field	Description
country	Specifies the country code to be used to retrieve the country-dependent information. If zero is given, the function uses the current country code. When the function returns, the country field receives the country code that the function actually used.
codepage	Specifies the code-page identifier to be used to retrieve information.

DosGetCtryInfo

The **COUNTRYINFO** structure pointed to by the *pctryiBuf* parameter 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;
```

Field	Description								
country	Specifies the country code.								
codepage	Specifies a reserved value. It must be zero.								
fsDateFmt	Specifies the date format. It can be one of the following values:								
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Month, day, year: mm/dd/yy</td></tr><tr><td>0x0001</td><td>Day, month, year: dd/mm/yy</td></tr><tr><td>0x0002</td><td>Year, month, day: yy/mm/dd</td></tr></table>	Value	Meaning	0x0000	Month, day, year: mm/dd/yy	0x0001	Day, month, year: dd/mm/yy	0x0002	Year, month, day: yy/mm/dd
Value	Meaning								
0x0000	Month, day, year: mm/dd/yy								
0x0001	Day, month, year: dd/mm/yy								
0x0002	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
0x0001	Currency indicator that follows a money value. If not given, the currency indicator precedes a money value.
0x0002	One space between the currency indicator and a money value. If not given, no space appears between the currency indicator and a money value.
0x0004	Currency indicator that replaces the decimal indicator. If this value is specified, 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. If it is not given, 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. It must be zero.

szDataSeparator[2] Specifies a data-list separator. It is a null-terminated string.

abReserved2[5] Specifies a reserved value. It must be zero.

Family API 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.

■ **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, so the current date and time are only as accurate as the most recent call to the **DosSetDateTime** function.

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

Parameter	Description
<i>pdateTime</i>	Points to the DATETIME structure that receives the date and time information. For a full description, see the following “Structures” section.

Return Value

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

Structures

The **DATETIME** structure pointed to by the *pdateTime* parameter 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;
```

Field	Description
hours	Specifies the current hour with values from 0 to 23.
minutes	Specifies the current minute with values from 0 to 59.
seconds	Specifies the current second with values from 0 to 59.
hundredths	Specifies the current hundredths of a second with values from 0 to 99.
day	Specifies the current day of the month with values from 1 to 31.
month	Specifies the current month of the year with values from 1 to 12.
year	Specifies the current year with values from 80 to 79 (1980 to 2079).

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, and negative for time zones east of Greenwich. For Eastern Standard Time, this field is 300—five hours after GMT.

weekday Specifies the current day of the week with values from 0 to 6 (Sunday equals zero).

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 under MS OS/2.

Example

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

```

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

```

See Also

DosGetInfoSeg, **DosSetDateTime**

■ **USHORT** **DosGetDBCSEv**(*cbBuf*, *pctrycCountry*, *pchBuf*)
USHORT *cbBuf*; length of buffer
PCOUNTRYCODE *pctrycCountry*; pointer to **COUNTRYCODE** structure
PCHAR *pchBuf*; pointer to buffer for DBCS information

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

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

Parameter	Description
<i>cbBuf</i>	Specifies the size in bytes of the buffer that receives the DBCS environmental vector.
<i>pctrycCountry</i>	Points to a COUNTRYCODE structure that contains the country code and code-page identifier to be used to retrieve the DBCS environmental vector. For a full description, see the following “Structures” section.

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

Return Value

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

ERROR_NLS_BAD_TYPE

An internal error occurred; the selector type does not exist.

ERROR_NLS_NO_COUNTRY_FILE

Cannot find the *country.sys* file.

ERROR_NLS_NO_CTRY_CODE

The country code was not found in the *country.sys* file.

ERROR_NLS_OPEN_FAILED

Cannot open the *country.sys* file.

ERROR_NLS_TABLE_TRUNCATED

The buffer is too small.

ERROR_NLS_TYPE_NOT_FOUND

An internal error occurred; the selector type is not in the file.

Structures

The **COUNTRYCODE** structure pointed to by the *pctrycCountry* parameter has the following form:

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

Field	Description
country	Specifies the country code to be used to retrieve the DBCS environmental vector. If zero is given, the function uses the current country code. When the function returns, the country field receives the country code that the function actually used.
codepage	Specifies the code-page identifier to be used.

Comments

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

The function copies the information from the *country.sys* file to the buffer pointed to by the *pchBuf* parameter. The first two bytes in the environmental vector specify the first and last values in the range for the DBCS lead-byte values. All subsequent pairs of bytes, except the last pair (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 information has a form that 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 the information, **DosGetDBCSEv** truncates the information. To avoid this, make sure the buffer is at least ten bytes long. Then verify that all information is copied by checking the last two bytes to make sure they are zeros. If the structure is larger than the information requires, the function fills any remaining bytes with zeros.

Family API 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**

- **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. The environment is one or more null-terminated strings that name and define the environment variables available to the current process. Environment variables are a way for the user to pass information to a program, for example, a list of directories that might contain data files to be used by the program.

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

Parameter	Description
<i>pselEnviron</i>	Points to the variable that receives the segment selector of the environment strings.
<i>pusOffsetCmd</i>	Points to the variable that receives the offset from the beginning of the segment identified by the <i>pselEnviron</i> parameter 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. The error value may be the following:

ERROR_INVALID_ACCESS

Comments

This function is typically used by library routines that need to determine the environment for the current process.

The environment begins in the first byte of the segment identified by the segment selector, the *pselEnviron* parameter. Each string in the environment has the following form:

name=value

The name of the environment variable is *name*; *value* is the variable's value. Both *name* and *value* can be any combination of characters, so it is entirely up to the program to determine how to interpret their meanings. Each string ends with a null character, and the entire environment ends with a null character.

The **DosGetEnv** function also retrieves an offset from the beginning of the segment to the first character in the command line used to start the process. This offset can be used to retrieve and process command-line arguments.

Example

This example calls the **DosGetEnv** function to retrieve the selector to the environment table, and the offset to the argument table within the environment table. It sets up the *pchEnviron* parameter to point to the beginning of the environment table, and the *pchArgument* parameter to point to the beginning of the argument table:

```
PCH pchEnviron, pchArgument;
SEL selEnviron;
USHORT usOffsetCmd;
DosGetEnv(&selEnviron, &usOffsetCmd);
pchEnviron = MAKEP(selEnviron, 0);
pchArgument = MAKEP(selEnviron, usOffsetCmd);
```

See Also

DosExecPgm

- **USHORT DosGetHugeShift(*pusShiftCount*)**
PUSHORT *pusShiftCount*; shift count returned

The **DosGetHugeShift** function retrieves the shift count used to compute the segment-selector offset for huge memory segments; namely, segments that are allocated by using the **DosAllocHuge** function. The shift count represents a multiple of 2, so the segment-selector offset is equal to the value 1 shifted left by the shift count. For example, the segment-selector offset is 8 if the shift count is 3.

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

Parameter	Description
<i>pusShiftCount</i>	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

- **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. The read-only information segments contain general information about the system and the process such as date, time, and the identifier of the current process. The global information segment is accessible to all processes. The local information segment is accessible to the current process only.

Parameter	Description
<i>pselGlobalSeg</i>	Points to the variable that receives the segment selector of the global information segment.
<i>pselLocalSeg</i>	Points to the variable that receives the segment selector of the local information segment.

Return Value

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

Structures

The global segment has an organization and content that is identical to the following structure:

```
typedef struct _GINFOSEG {
    ULONG    time;
    ULONG    msec;
    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];
} GINFOSEG;
```

Field	Description
time	Specifies the time in seconds from January 1, 1970.
msec	Specifies the current system time in milliseconds.
hour	Specifies the current hour with values from 0 to 23.

minutes	Specifies the current minute with values from 0 to 59.
seconds	Specifies the current second with values from 0 to 59.
hundredths	Specifies the current hundredths of a second with values from 0 to 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, and negative for time zones east of Greenwich. For Eastern Standard Time, this field is 300—five hours after GMT. If this field is -1, the time zone is undefined.
cusecTimerInterval	Specifies the timer interval in milliseconds (0.0001 seconds).
day	Specifies the current day of the month with values from 1 to 31.
month	Specifies the current month of the year with values from 1 to 12.
year	Specifies the current year with values from 80 to 79 (1980 to 2079).
weekday	Specifies the current day of the week with values from 0 to 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 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]** Each bit corresponds to a system-trace major code from 0x0000 to 0x00FF. The most significant bit (left-most) 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.

The local segment has an organization and content that is identical to the following structure:

```
typedef struct _LINFOSEG {
    PID      pidCurrent;
    PID      pidParent;
    USHORT   prtyCurrent;
    TID      tidCurrent;
    USHORT   sgCurrent;
    USHORT   sgSub;
    BOOL     fForeground;
} LINFOSEG;
```

Field	Description
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.
sgSub	Specifies the subscreen group.
fForeground	Specifies that the current process is in foreground.

Example

This example calls the **DosGetInfoSeg** function to get the selector of a system global segment. It then converts the selector into a structure pointer, and executes a **while** loop until the hour stored in the global selector is 10:

```
SEL selGlobalSeg, selLocalSeg;
GINFOSEG FAR *pgisInfo;
DosGetInfoSeg(&selGlobalSeg, &selLocalSeg);
pgisInfo = MAKEPGINFOSEG(selGlobalSeg);
while (pgisInfo->hour != 10);          /* Do until 10:00 a.m. */
```


See Also

DosGetDateTime

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

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

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

Parameter	Description
<i>pbMachineMode</i>	Points to the variable that receives the machine mode. If this parameter is 0x0000, the current mode is real mode, 808x or 80286. If this parameter is 0x0001, the current mode is protected mode, 80286.

Return Value

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

Comments

Not all MS OS/2 functions are available in real mode. Since bound programs can run in either mode, **DosGetMachineMode** lets a program determine whether it is in protected or real mode. Consequently, the program can 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 Chapter 2, "Overview."

Example

This example calls the **DosGetMachineMode** function and displays whether the current process is running in protected or real mode:

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

- **USHORT DosGetMessage**(*ppchVTable, usVCount, pchBuf, cbBuf, usMsgNo, pszFileName, pcbMsg*)

PCHAR FAR * <i>ppchVTable</i> ;	pointer to table of character pointers
USHORT <i>usVCount</i> ;	number of pointers in table
PCHAR <i>pchBuf</i> ;	pointer to buffer for return message
USHORT <i>cbBuf</i> ;	length of buffer
USHORT <i>usMsgNo</i> ;	message number to retrieve
PSZ <i>pszFileName</i> ;	name of file containing message
PUSHORT <i>pcbMsg</i> ;	length of returned message

The **DosGetMessage** function retrieves a message from the specified system message file. It 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.

Parameter	Description
<i>ppchVTable</i>	Points to a table of substitution strings. Each entry in the table points to a null-terminated string to be inserted into the message. Up to nine strings can be given.
<i>usVCount</i>	Specifies the number of strings in the table. It can be any value from 0 to 9. If this parameter is zero, the <i>ppchVTable</i> parameter is ignored. If it is greater than 9, DosGetMessage returns an error indicating that <i>usVCount</i> is out of range.
<i>pchBuf</i>	Points to the buffer that receives the requested message.
<i>cbBuf</i>	Specifies the length in bytes of the buffer.
<i>usMsgNo</i>	Specifies the message number for the requested message.
<i>pszFileName</i>	Points to a null-terminated string that specifies the MS OS/2 path and filename of the message file that contains the message.
<i>pcbMsg</i>	Points to the variable that receives the number of bytes actually copied to the buffer.

Return Value

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

ERROR_FILE_NOT_FOUND
The file was not found.

ERROR_MR_INV_IVCOUNT

The insertion-variable count is invalid.

ERROR_MR_INV_MSGF_FORMAT

The message-file format is invalid.

ERROR_MR_MID_NOT_FOUND

The message-identifier number was not found.

ERROR_MR_MSG_TOO_LONG

The message is too long for the buffer.

ERROR_MR_UN_ACC_MSGF

Unable to access the message file.

ERROR_MR_UN_PERFORM

Unable to perform the requested function.

Comments

The *usMsgNo* parameter identifies the requested message; the *pszFileName* parameter identifies the message file that contains the message.

To retrieve the requested message, the **DosGetMessage** function first searches the process's message segment, if there is one. If it cannot find the message, the function then searches the specified file for the message. If no drive or path is specified in the filename, **DosGetMessage** searches the system root directory for the file. It then searches the current directory on the current drive, if necessary. 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 **DosGetMessage** finds a message, it copies the message to the buffer pointed to by the *pchBuf* parameter. As it copies the messages, **DosGetMessage** replaces any symbol in the form *%x* (where *x* is a digit from 1 to 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 first string in the table. If there is no corresponding string in the table, **DosGetMessage** copies the *%x* sequence, unchanged, to the buffer.

There is no guarantee that the *%x* symbols used in a message are 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, **DosGetMessage** truncates it and returns an error code.

If **DosGetMessage** cannot retrieve a message because of a direct-access-storage-device (DASD) hard error or a file-not-found condition, it places one of the following default messages in the buffer:

“Unable to format the system message file”

This message occurs when an invalid parameter is specified, for example, an invalid *usMsgNo* or an invalid *usVCount*.

“Unable to read the system message file”

This message occurs when the function cannot read the system message file, for example, due to a DASD error or an invalid message file format.

“Unable to find the system message file”

This message occurs when the function cannot find the system message file.

The **DosGetMessage** function retrieves messages that have been previously prepared using the **mkmsgf** utility to create a message file. It also retrieves messages that have been added to the message segment of the program’s executable file using the **msgbind** utility. Whether **DosGetMessage** retrieves messages from the message segment or from the message file is transparent to the process that calls **DosGetMessage**. 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*.

Family API Restrictions

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

DosInsMessage, **DosPutMessage**

■ **USHORT DosGetModHandle(pszModName, phMod)**
PSZ pszModName; name of module
PHMODULE phMod; pointer to variable for module handle

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

Parameter	Description
<i>pszModName</i>	Points to a null-terminated string. The string specifies the MS OS/2 filename of the module. The <i>.dll</i> filename extension is used for dynamic-link libraries.
<i>phMod</i>	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. The error value may be one of the following:

ERROR_INTERRUPT

ERROR_MOD_NOT_FOUND

Comments

The module name must match the name of the module already loaded. Otherwise, an error value is returned.

See Also

DosFreeModule, **DosGetModName**, **DosLoadModule**

■ **USHORT** **DosGetModName**(*hmod*, *cbBuf*, *pchBuf*)
HMODULE *hmod*; module handle
USHORT *cbBuf*; buffer length
PCHAR *pchBuf*; pointer to buffer for module name

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

Parameter	Description
<i>hmod</i>	Identifies the dynamic-link module. The handle must have been previously created by using the DosLoadModule function.
<i>cbBuf</i>	Specifies the maximum length in bytes of the buffer that receives the name.
<i>pchBuf</i>	Points to the buffer that receives the module's filename.

Return Value

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

ERROR_BAD_LENGTH

ERROR_INTERRUPT

ERROR_INVALID_HANDLE

Comments

DosGetModName returns an error if there is not enough room in the buffer for the filename.

The module handle for the current process can be retrieved from the **DI** register on entry to a module or upon initialization entry to a dynamic-link module.

See Also

DosFreeModule, **DosLoadModule**, **DosMonOpen**

■ **USHORT** **DosGetPid**(*ppidiInfo*)
PPIDINFO *ppidiInfo*; pointer to buffer for process information

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

Parameter	Description
<i>ppidiInfo</i>	Points to the PIDINFO structure that receives the process identifiers. For a full description, see the following “Structures” section.

Return Value

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

Structures

The **PIDINFO** structure pointed to by the *ppidiInfo* parameter has the following form:

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

Field	Description
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

DosExecPgm

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

The **DosGetProcAddress** function retrieves the address of the desired procedure within the given dynamic-link module. The procedure address can then be used to call the procedure.

Parameter	Description
<i>hmod</i>	Identifies the dynamic-link module. The handle must have been previously created by using the DosMon-Open function.
<i>pszProcName</i>	Points to a null-terminated string. The string specifies the name of the procedure to be retrieved. If the 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.
<i>ppfnProcAddress</i>	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. The error value may be one of the following:

ERROR_INTERRUPT

ERROR_INVALID_HANDLE

ERROR_PROC_NOT_FOUND

Comments

Although the **DosGetProcAddress** 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 DOSCALLS by using a procedure name, **DosGetProcAddress** returns an error. Check the *doscalls.lib* library for a list of DOSCALLS procedure names and corresponding ordinal numbers.

Example

This example calls the **DosLoadModule** function to load the dynamic-link module *qhdl.dll*. It then calls the **DosGetProcAddress** function to retrieve the address of the BOXMESSAGE function defined in the module. After calling the function defined in the dynamic-link module, it calls **DosFreeModule** to free the dynamic-link module. This example assumes the existence of *qhdl.dll* in a directory defined by the *libpath* parameter of *config.sys*, and that it 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), "qhdl", &hmod);
DosGetProcAddress(hmod, /* handle of module */
    "BOXMESSAGE", /* name of function */
    &pfnBoxMsg); /* variable for function address */
pfnBoxMsg("Hello World", 0x30, 1, 0, 0);
DosFreeModule(hmod);
```

See Also

DosFreeModule, **DosGetModName**, **DosLoadModule**,
DosMonOpen

- **USHORT DosGetPrtty(usScope, pusPriority, pid)**
USHORT usScope; Indicates scope of the query
PUSHORT pusPriority; address of priority
USHORT pid; process or thread identifier

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

Parameter	Description
<i>usScope</i>	Specifies whether to retrieve the priority for a thread in the current process or in some other process. If the <i>usScope</i> parameter is <code>PRTYS_PROCESS</code> , the function retrieves the priority of thread 1 for the process specified by the <i>pid</i> parameter. If it is <code>PRTYS_THREAD</code> , the function retrieves the priority of the thread specified by the <i>pid</i> parameter.
<i>pusPriority</i>	Points to the variable that receives the scheduling priority of the given thread. The high-order byte is set to the priority class; the low-order byte is set to the level.
<i>pid</i>	Specifies a process or thread identifier, depending on the value of the <i>usScope</i> parameter. If <i>usScope</i> is <code>PRTYS_PROCESS</code> , <i>pid</i> specifies a process identifier. If <i>usScope</i> is <code>PRTYS_THREAD</code> , <i>pid</i> specifies a thread identifier. If the <i>pid</i> parameter is 0x0000, the 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. The error value may be one of the following:

ERROR_INVALID_PROCID

The process identifier is invalid or nonexistent.

ERROR_INVALID_SCOPE

The *usScope* parameter must indicate either a thread or process.

ERROR_INVALID_THREADID

The thread identifier is invalid or nonexistent.

Comments

For more information about priorities, see the **DosSetPrty** function.

If the *usScope* parameter is equal to `PRTYS_PROCESS` (process identifier), the priority of the first thread of the process is returned. If that thread has terminated, **DosGetPrty** returns an error value.

See Also

DosSetPrty

- **USHORT DosGetSeg(*sel*)**
SEL *sel*; selector to access

The **DosGetSeg** function secures access to the shared memory segment identified by the segment selector specified by the *sel* parameter. 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 has secured access by using the **DosGetSeg** function.

Parameter	Description
<i>sel</i>	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

The **DosGetSeg** function applies only to the shared memory segment that was created by using the **DosAllocSeg** function with the *fAlloc* parameter set to **SEG_GETTABLE**.

See Also

DosAllocSeg, DosGiveSeg

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

The **DosGetShrSeg** function retrieves a selector to the shared memory segment pointed to by the *pszName* parameter. The shared segment must have been previously allocated 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 access the shared memory segment.

Parameter	Description
<i>pszName</i>	Points to a null-terminated string. The string identifies the shared memory segment and must have the following form: \sharemem\ <i>pszName</i>

The string name, *pszName*, 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 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. The error value may be one of the following:

`ERROR_FILE_NOT_FOUND`

The name is not found in the shared-segment tables.

`ERROR_INVALID_HANDLE`

The share name is invalid.

`ERROR_TOO_MANY_OPEN_FILES`

Comments

Although the new selector and the selector originally created by the **DosAllocShrSeg** function may not be the same, they both identify the same shared memory segment.

See Also

DosAllocShrSeg, **DosFreeSeg**

■ USHORT DosGetVersion(*pusVersion*)

PUSHORT *pusVersion*; points to variable for version number

The **DosGetVersion** function retrieves the operating system's version number.

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

Parameter	Description
<i>pusVersion</i>	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.

■ USHORT DosGiveSeg(*sel*, *pidProcess*, *pselRecipient*)

SEL <i>sel</i> ;	caller's selector
PID <i>pidProcess</i> ;	pointer to variable with process ID of recipient
PSEL <i>pselRecipient</i> ;	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 the other process by using some form of interprocess communication.

Parameter	Description
<i>sel</i>	Specifies the segment selector of the shared memory segment.
<i>pidProcess</i>	Specifies the process identifier of the process that receives access to the shared memory segment.
<i>pselRecipient</i>	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. The error value 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.

The **DosGiveSeg** function applies only to the shared memory segment created by using the **DosAllocSeg** function with the *fAlloc* parameter set to `SEG_GETTABLE`.

See Also

DosAllocSeg

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

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

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

Parameter	Description
<i>fDisable</i>	Specifies whether to disable or enable signals intended for the current process. If this parameter is HLDSIG_ENABLE , the function enables signals and restores the state to the same as before the last call to DosHoldSignal . If it is HLDSIG_DISABLE , the function disables signals.

Return Value

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

ERROR_INVALID_FUNCTION

Comments

If the *fDisable* parameter is **HLDSIG_DISABLE**, the function postpones all signal processing until **DosHoldSignal** is called again with *fDisable* set to **HLDSIG_ENABLE**. Any signals that occur while processing is disabled are recognized but not accepted until processing is enabled.

Signals should be held for as short a period of time as possible, being released and reheld if necessary. Their use and guidelines for proper use are similar to those of their hardware-interrupt counterparts, the **cli** and **sti** instructions.

Requests to disable and enable signal processing are cumulative. This means two requests to disable processing must be followed by two requests to enable before processing is enabled. If the request count is too large or too small, or if *fDisable* is not **HLDSIG_ENABLE** or **HLDSIG_DISABLE**, **DosHoldSignal** returns an error value.

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 critical sections or in temporarily reserved resources.

Family API Restrictions

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

- Only the signal interrupt (SIGINTR) and signal break (SIGBREAK) signals are recognized.

Example

This example calls the **DosHoldSignal** function to disable signals and calls the **DosEnterCritSec** function to enter a critical section of code (all other threads are stopped). When the critical section is done, 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**

- **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; length of input message
PCHAR pchBuf; pointer to buffer for updated message
USHORT cbBuf; length of buffer
PUSHORT pcbMsg; pointer to variable for length of message

The **DosInsMessage** function copies a message from the *pszMsg* parameter to a buffer pointed to by the *pchBuf* parameter.

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

Parameter	Description
<i>ppchVTable</i>	Points to a table of substitution strings. Each entry in the table points to a null-terminated string to be inserted into the message. Up to nine strings can be given.

<i>usVCount</i>	Specifies the number of strings in the table. It can be any value from 0 to 9. If this parameter is zero, the <i>ppchVTable</i> parameter is ignored. If the parameter is greater than 9, the function returns an error value indicating that the <i>usVCount</i> parameter is out of range.
<i>pszMsg</i>	Points to a null-terminated string that specifies the message to be processed.
<i>cbMsg</i>	Specifies the length in bytes of the input message.
<i>pchBuf</i>	Points to the buffer that receives the requested message.
<i>cbBuf</i>	Specifies the length in bytes of the buffer.
<i>pcbMsg</i>	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. The error value may be one of the following:

ERROR_MR_INV_IVCOUNT

The insertion-variable count is invalid.

ERROR_MR_MSG_TOO_LONG

The message is too long for the buffer.

Comments

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

Unlike the **DosGetMessage** function, **DosInsMessage** does not retrieve a message. It is most useful when messages are loaded early before the insertion text strings are known.

Family API 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

- **USHORT DosKillProcess**(*fScope*, *pidProcess*)
USHORT *fScope*; children/process-only flag
PID *pidProcess*; process ID of process to be ended

The **DosKillProcess** function terminates the specified process, and optionally, all child processes that belong to the specified process. Any subsequent request for the process's termination code (for example, by calling the **DosCWait** function) returns the "abnormal termination" code.

Parameter	Description
<i>fScope</i>	Specifies whether to terminate any existing child processes that belong to the process that is terminated. If this parameter is DKP_PROCESS , the function terminates the specified process and all of its child processes. If it is DKP_PROCESSTREE , the function terminates the specified processes only.
<i>pidProcess</i>	Specifies the process identifier of the process to be terminated.

Return Value

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

ERROR_INVALID_PROCID

Comments

A process can intercept the termination request generated by the **DosKillProcess** function by creating a signal handler using the **DosSetSigHandler** function. In such a case, 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 file buffers and closing all handles opened by the process.

The process must flush non-system file buffers before terminating. A non-system file buffer is a file buffer that is not managed by MS OS/2; for example, the character buffers managed by the C-language run-time library are non-system buffers. MS OS/2 does not flush these buffers as part of its termination sequence.

Example

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

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

See Also

DosCWait, **DosExit**, **DosSetSigHandler**

■ USHORT DosLoadModule(*pszFailName*, *cbFileName*, *pszModName*, *phModule*)

PSZ <i>pszFailName</i> ;	pointer to buffer for failure name
USHORT <i>cbFileName</i> ;	length of failure-name buffer
PSZ <i>pszModName</i> ;	pointer to module name
PHMODULE <i>phmod</i> ;	pointer to variable for module handle

The **DosLoadModule** function loads a dynamic-link module and returns a handle for the module. A dynamic-link module (also called a dynamic-link library) typically contains procedures that the process may link to and call. The module handle is used to retrieve the entry addresses of procedures in the module as well as to retrieve information about the module.

Parameter	Description
<i>pszFailName</i>	Points to the buffer that receives a null-terminated string. The function copies a string to the buffer only if it fails to load the module. The string identifies the dynamic-link module responsible for the failure. This may be a module other than the one specified if the specified module itself links to other dynamic-link modules.
<i>cbFileName</i>	Specifies the length in bytes of the name buffer.
<i>pszModName</i>	Points to a null-terminated string. The 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 <i>.dll</i> 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. The error value may be one of the following:

ERROR_BAD_FORMAT

ERROR_FILE_NOT_FOUND

ERROR_INTERRUPT

ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosLoadModule** function loads the module only if it has not been previously loaded. Only preload segments of the module are loaded immediately; demand-load segments are loaded as needed, for example, when the process calls a procedure in the module. If the module was previously loaded, the function increases the module's reference count by one. The reference count indicates how many processes have loaded the module; the module remains loaded as long as the reference count is not zero.

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 *qhdl.dll*. It then calls the **DosGetProcAddress** function to retrieve the address of the BOXMESSAGE function defined in the module. After calling the function defined in the dynamic-link library, it calls **DosFreeModule** to free the dynamic-link module. This example assumes the existence of *qhdl.dll* in a directory defined by the *libpath* parameter of *config.sys*, and that it 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 */
              "qhdl",               /* module name */
              &hmod);              /* address of handle */
DosGetProcAddress(hmod, "BOXMESSAGE", &pfnBoxMsg);
pfnBoxMsg("Hello World", 0x30, 1, 0, 0, FALSE);
DosFreeModule(hmod);
```

See Also

DosExecPgm, DosFreeModule, DosGetModName, DosGetProcAddr

■ **USHORT DosLockSeg(*sel*)**
SEL *sel*; selector to lock

The **DosLockSeg** function locks a discardable segment in memory. The process can then access the segment without the contents of the segment being discarded; that is, a segment cannot be discarded while it is locked. A locked segment cannot be discarded until it is unlocked by using the **DosUnlockSeg** function.

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

Parameter	Description
<i>sel</i>	Specifies the selector of the segment to be locked.

Return Value

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

Comments

The **DosLockSeg** function 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 still move and swap a locked segment as needed.

The **DosLockSeg** and **DosUnlockSeg** functions may be nested. For example, 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, DosReallocSeg, DosUnlockSeg

- **USHORT DosMakePipe**(*phfRead*, *phfWrite*, *cbPipe*)
PHFILE *phfRead*; pointer to variable for read handle
PHFILE *phfWrite*; pointer to variable for write handle
USHORT *cbPipe*; bytes reserved for pipe

The **DosMakePipe** function creates a pipe. A pipe is temporary method of storage that a process can write to and read from as if it were a file. The function creates the pipe, assigning the specified pipe size to the storage buffer, and creates handles that the process can use in subsequent calls to the **DosRead** and **DosWrite** functions to read from and write to the buffer.

Parameter	Description
<i>phfRead</i>	Points to the variable that receives the read handle for the pipe.
<i>phfWrite</i>	Points to the variable that receives the write handle for the pipe.
<i>cbPipe</i>	Specifies the size in bytes to allocate for the storage buffer for this pipe. It can be any value from 0 to 65,504 (the maximum size is 65,536 minus the size of the pipe header, 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. The error value 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.

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

MS OS/2 prevents a pipe from overflowing by temporarily blocking a call to any **DosWrite** function that attempts to write more data to the pipe than can 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

■ **USHORT DosMemAvail(*pulAvailMem*)**
PULONG *pulAvailMem*; available memory

The **DosMemAvail** function retrieves the size of the largest block of free memory currently available. 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.

Parameter	Description
<i>pulAvailMem</i>	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 be allocating and freeing memory, the largest free block can be expected to change at any time.

The returned value may be valid only at the moment **DosMemAvail** is called.

- **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 not valid, the function returns an error value.

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

Parameter	Description
<i>pszDirName</i>	Points to a null-terminated string. The string must be a valid MS OS/2 directory name.
<i>ulReserved</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_ACCESS_DENIED

This directory already exists or a file with this name exists.

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 cannot create the subdirectory:

```
if (DosMkdir("abc", OL))
    VioWrtTTY("Can't open directory\n\r", 22, 0);
else {
```

See Also

DosRmdir

- **USHORT** **DosMonClose**(*hmon*)
HMONITOR *hmon*; handle from **DosMonOpen**

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

Parameter	Description
<i>hmon</i>	Identifies the monitor to be closed. The handle must have been previously created by using the DosMonOpen function.

Return Value

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

ERROR_MON_INVALID_HANDLE
The monitor handle is invalid.

See Also

DosMonOpen, **DosMonRead**, **DosMonReg**, **DosMonWrite**

- **USHORT DosMonOpen(*pszDevName*, *phmon*)**
PSZ *pszDevName*; points to device name
PHMONITOR *phmon*; points to variable for 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.

Parameter	Description
<i>pszDevName</i>	Points to a null-terminated string. The string specifies the name of the device for which the monitor is to be opened.
<i>phmon</i>	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. The error value may be one of the following:

ERROR_MON_INVALID_DEVNAME
The device-name string is invalid.

ERROR_NOT_ENOUGH_MEMORY
There is insufficient memory to open the monitor.

See Also

DosMonClose, DosMonRead, DosMonWrite

■ **USHORT DosMonRead**(*pbInBuffer*, *fNoWait*, *pbDataBuf*, *pcbDataBuf*)
PBYTE *pbInBuffer*; pointer to buffer for monitor input
USHORT *fNoWait*; 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 the buffer pointed to by the *pbDataBuf* parameter.

Parameter	Description
<i>pbInBuffer</i>	Points to the monitor input buffer. This handle must have been previously registered by using the DosMonReg function.
<i>fNoWait</i>	Specifies whether the function should wait for input. If this parameter is DCWW_NOWAIT , the function waits until input is ready. If it is DCWW_WAIT , the function returns immediately if no input is ready.
<i>pbDataBuf</i>	Points to the buffer that receives the data records.
<i>pcbDataBuf</i>	Points to the variable that contains the size in bytes of the buffer that receives the data records. When the 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. The error value may be one of the following:

ERROR_MON_BUFFER_EMPTY

No data is present in the input buffer.

ERROR_MON_BUFFER_TOO_SMALL

The buffer is too small; the buffer lengths must each be greater than or equal to 64 bytes.

ERROR_MON_INVALID_PARMS

One of the parameters is invalid.

Comments

Since device monitors are part of the data flow path through a device driver, they must respond rapidly so that they do not delay input and output (I/O). This is especially important in the case of keyboard monitors. A monitor process should be written so that the thread (or threads) that reads and writes the monitor data runs at a high priority. These threads should never perform operations, such as I/O or semaphore waits, that might delay them. The monitor process can have another thread (or threads), running at normal priority to handle such operations.

See Also

DosMonClose, DosMonOpen, DosMonReg, DosMonWrite

■ **USHORT DosMonReg**(*hmon, pbInBuf, pbOutBuf, fPosition, usIndex*)
HMONITOR *hmon*; handle from **DosMonOpen**
PBYTE *pbInBuf*; pointer to monitor-input buffer
PBYTE *pbOutBuf*; pointer to monitor-output buffer
USHORT *fPosition*; position flag
USHORT *usIndex*; index

The **DosMonReg** function registers a monitor. It also creates the input- and output-buffer structures used to monitor the input and output stream of a given device.

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.

The **DosMonReg** function will format the buffer as needed.

Parameter	Description
<i>hmon</i>	Identifies the monitor to be registered. The handle must have been previously created by using the DosMonOpen function.
<i>pbInBuf</i>	Points to the input-buffer structure. This buffer receives data from the previous monitor. For a full description, see the following "Structures" section.
<i>pbOutBuf</i>	Points to the output-buffer structure. This buffer receives data for the next monitor in the chain. For a full description, see the following "Structures" section.

fPosition Specifies the position of the monitor in the input and output chain. It can be one of the following values:

Value	Meaning
0x0000	Place the monitor anywhere in the chain. The function places the monitor at a convenient location in the chain.
0x0001	Place the monitor at the beginning of the chain. The function places the specified monitor in front of any other monitors already in the chain.
0x0002	Place the monitor at the end of the chain. The function places the specified monitor after any other monitors already in the chain.

usIndex Specifies a device-specific value. For the keyboard, it specifies the identifier for the screen group to be monitored. 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 called the **KbdCharIn** last.

Return Value

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

ERROR_MON_BUFFER_TOO_SMALL

The buffer is too small; the buffer lengths must each be greater than or equal to 64 bytes.

ERROR_MON_INVALID_HANDLE

The monitor handle is invalid.

ERROR_MON_INVALID_PARMS

One of the parameters is invalid.

ERROR_NOT_ENOUGH_MEMORY

There is insufficient memory to register the monitor.

Structures

The input-buffer structure, pointed to by the *pbInBuf* parameter, must have a structure and content that is identical to the following structure:

```
struct {
    USHORT length;
    UCHAR buffer[18];
    .
    .
};
```

Field	Description
length	Specifies the length of the structure in bytes. The structure must be at least 64 bytes, although 128 bytes is the recommended length.
buffer[18]	Specifies a reserved space.

The output-buffer structure, pointed to by the *pbOutBuf* parameter, must have a structure and content that is identical to the following structure:

```
struct {
    USHORT length;
    UCHAR buffer[18];
    .
    .
};
```

Field	Description
length	Specifies the length of the structure in bytes. The structure must be at least 64 bytes, although 128 bytes is the recommended length.
buffer[18]	Specifies a reserved space.

The input- and output-buffer structures must be in the same segment.

See Also

DosMonClose, DosMonOpen, DosMonRead, 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 the output stream for a device. The output-buffer structure identifies the device that receives the data records.

Parameter	Description
<i>pbOutBuf</i>	Points to the output-buffer structure for the monitor. It must have been previously registered by using the DosMonReg function.
<i>pbDataBuf</i>	Points to the buffer that contains the data records to be inserted into the output stream for the device.
<i>cbDataBuf</i>	Specifies the number of bytes of data records in the buffer pointed to by the <i>pbDataBuf</i> parameter.

Return Value

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

ERROR_MON_DATA_TOO_LARGE

The data record length is greater than the output buffer.

ERROR_MON_INVALID_PARMS

One of the parameters is invalid.

Comments

Since device monitors are part of the data flow path through a device driver, they must respond rapidly so that input and output (I/O) is not delayed. This rapid response is especially important in the case of keyboard monitors. A monitor process should be written so that the thread (or threads) that actually reads and writes the monitor data runs at a high priority. These threads should never perform operations, such as I/O or semaphore waits, that might delay them. The monitor process can have another thread (or threads) running at normal priorities to handle such operations.

See Also

DosMonClose, DosMonOpen, DosMonRead, DosMonReg

- **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 the file specified by the *pszOldName* parameter to the new directory and/or filename specified by the *pszNewName* parameter. The function is typically used to rename an

existing file; that is, it moves the file to a new filename location in the same directory. It can also be used to move a file to a new directory while preserving the existing filename.

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

Parameter	Description
<i>pszOldName</i>	Points to a null-terminated string. The string specifies the current filename of the file to be moved. It must be a valid MS OS/2 filename.
<i>pszNewName</i>	Points to a null-terminated string. The string specifies the new directory and filename of the file to be moved. It must be a valid MS OS/2 filename.
<i>ulReserved</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_ACCESS_DENIED

Cannot move the directory.

ERROR_DRIVE_LOCKED

ERROR_FILE_NOT_FOUND

ERROR_NOT_DOS_DISK

ERROR_NOT_SAME_DEVICE

ERROR_PATH_NOT_FOUND

Either the old path or the new path cannot be found.

ERROR_SHARING_BUFFER_EXCEEDED

ERROR_SHARING_VIOLATION

One of the files is open.

Comments

The **DosMove** function cannot move a file from one drive to another, so 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 then to rename it *xyz*. This does not copy the file, but optionally changes the subdirectory that the filename appears in and optionally changes the filename itself. Because no copying is done, both the old and new filenames must be on the same drive:

```
DosMove("abc",          /* old filename and path */
        "\\xyz",        /* new filename and path */
        OL);            /* Reserved                */
```

See Also

DosDelete, **DosSelectDisk**

- **USHORT DosMuxSemWait**(*pusSemIndex*, *pvoidSemList*, *lTimeOut*)
PUSHORT *pusSemIndex*; pointer to variable for cleared semaphore
PVOID *pvoidSemList*; pointer to 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 *pvoidSemList* parameter. If any of the semaphores 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.

Parameter	Description
<i>pusSemIndex</i>	Points to the variable that receives the index number of the most recently cleared semaphore.
<i>pvoidSemList</i>	Points to a semaphore list that defines the semaphores to be cleared. The semaphore list consists of one or more semaphore handles. For a full description, see the following "Structures" section.
<i>lTimeOut</i>	<p>Specifies an elapsed time in milliseconds to wait for the semaphores to clear.</p> <p>If the <i>lTimeOut</i> parameter is 0xFFFFFFFF, the function waits indefinitely for a semaphore to clear. If it is 0x00000000, the function returns immediately if no semaphore is clear. Any other value is considered to be the amount of time to wait.</p>

Return Value

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

The system limit of 100 entries has been reached.

Structures

The semaphore list has an organization and content that is identical to the following structures:

```
typedef struct _MUXSEM {
    USHORT zero;
    HSEM hsem;
} MUXSEM;

typedef struct _MUXSEMLIST {
    USHORT cmxs;
    MUXSEM amxs[16];
} MUXSEMLIST;
```

Field	Description
zero	Specifies a reserved value. It must be zero.
hsem	Specifies the semaphore handle. It must have been previously created by using the DosCreateSem or DosOpenSem function.
cmxs	Specifies the number of semaphores in the list.
amxs[16]	Specifies an array of MUXSEM structures.

The structure contains one **reserved** and **semhandle** field pair for each semaphore in the list. The structure may contain up to 16 semaphores.

Comments

Although declared with **PVOID** type, the second parameter of the **DosMuxSemWait** function must point to a **MUXSEMLIST** structure. You can create the structure 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. The macro actually 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 quite 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, then immediately set it again. Threads that waited for that semaphore will return from **DosMuxSemWait**. Those 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 *IndexNum* parameter to find out which semaphore is cleared, and if it is semaphore 1, the example sets it:


```

DEFINEMUXSEMLIST (MuxList, 2)          /* Creates structure array */
USHORT usSemIndex;
MuxList.cmxs = 2;
DosCreateSem(1, &MuxList.amxs[0].hsem,
"\SEM\TIMER0.SEM");
DosCreateSem(1, &MuxList.amxs[1].hsem,
"\SEM\TIMER1.SEM");
.
.
DosMuxSemWait(&usSemIndex, &MuxList, 5000L);
if (usSemIndex == 1) {
    DosSemSet (MuxList.amxs[1].hsem);
}

```

See Also

DosSemRequest, DosSemSetWait, DosSemWait

■ **USHORT DosNewSize**(*hf*, *ulNewSize*)
HFILE *hf*; handle to file
ULONG *ulNewSize*; file's new size

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 new bytes is undefined.

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

P rameter	Description
<i>hf</i>	Identifies the file to be changed. The handle must have been previously created by using the DosOpen function.
<i>ulNewSize</i>	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. The error value may be one of the following:

ERROR_ACCESS_DENIED
This file has been opened as a read-only file.

ERROR_DISK_FULL

ERROR_INVALID_HANDLE

ERROR_INVALID_PARAMETER

The error is caused by an invalid size.

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, and 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 existed and was larger 100 bytes, it will be truncated to 100 bytes. If the file was smaller than 100 bytes, or if it was created by using the **DosOpen** function, it will be expanded to 100 bytes:

```
HFILE hf;  
USHORT usAction;  
DosOpen("abc", &hf, &usAction, OL, O, 0x11, 0x42, OL);  
DosNewSize(hf, 100L);
```

See Also

DosOpen, DosQFileInfo, DosSetFileMode

- **USHORT DosOpen**(*pszFileName*, *phf*, *pusAction*, *ulFileSize*,
usAttribute, *fOpenFlags*, *fOpenMode*, *ulReserved*)
- | | |
|------------------------------------|---|
| PSZ <i>pszFileName</i> ; | pointer to filename |
| PHFILE <i>phf</i> ; | new pointer to variable for file's handle |
| PUSHORT <i>pusAction</i> ; | pointer to variable for action taken |
| ULONG <i>ulFileSize</i> ; | initial allocation size |
| USHORT <i>usAttribute</i> ; | file attribute |
| USHORT <i>fOpenFlags</i> ; | open function type |
| USHORT <i>fOpenMode</i> ; | open mode of file |
| ULONG <i>ulReserved</i> ; | Must be zero |

The **DosOpen** function opens an existing file or creates a new file. It returns a handle that can be used to read from and write to the file, as well as retrieve information about the file.

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

Parameter	Description												
<i>pszFileName</i>	Points to a null-terminated string. The string 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.												
<i>phf</i>	Points to a variable that receives the handle to the opened file.												
<i>pusAction</i>	Specifies the action taken as a result of the DosOpen function. If DosOpen fails, the action-taken value has no meaning. Otherwise, it is one of the following values:												
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>File already existed.</td></tr> <tr> <td>0x0002</td><td>File was created.</td></tr> <tr> <td>0x0003</td><td>File existed and was truncated.</td></tr> </table>	Value	Meaning	0x0001	File already existed.	0x0002	File was created.	0x0003	File existed and was truncated.				
Value	Meaning												
0x0001	File already existed.												
0x0002	File was created.												
0x0003	File existed and was truncated.												
<i>ulFileSize</i>	Specifies the file's new size in bytes. It applies only if the file is created or truncated. The size has no effect on a file opened for reading.												
<i>usAttribute</i>	Specifies the file attributes. It can be a combination of the following values:												
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>Normal file</td></tr> <tr> <td>0x0001</td><td>Read-only file</td></tr> <tr> <td>0x0002</td><td>Hidden file</td></tr> <tr> <td>0x0004</td><td>System file</td></tr> <tr> <td>0x0020</td><td>Archived file</td></tr> </table> <p>File attributes apply only if the file is created.</p>	Value	Meaning	0x0000	Normal file	0x0001	Read-only file	0x0002	Hidden file	0x0004	System file	0x0020	Archived file
Value	Meaning												
0x0000	Normal file												
0x0001	Read-only file												
0x0002	Hidden file												
0x0004	System file												
0x0020	Archived file												

fOpenFlags Specifies the action to take if the file does or does not exist. It can be a combination of the following values:

Value	Meaning
0x0000	Fail if the file exists.
0x0001	Open the file.
0x0002	Truncate the file.
0x0010	Create the file if it does not exist.

fOpenMode Specifies the modes with which to open the file. For a full description, see Table 3.1.

ulReserved Specifies a reserved value. It must be zero.

Return Value

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

ERROR_ACCESS_DENIED

An attempt was made to open a directory or open a read-only file with write access.

ERROR_CANNOT_MAKE

There is no room to create a directory entry.

ERROR_DISK_FULL

ERROR_DRIVE_LOCKED

ERROR_FILE_NOT_FOUND

The filename is invalid.

ERROR_INVALID_ACCESS

The access mode field is invalid.

ERROR_INVALID_PARAMETER

An invalid *fOpenFlags* parameter was specified.

ERROR_NOT_DOS_DISK

ERROR_OPEN_FAILED

ERROR_PATH_NOT_FOUND

An invalid path was specified.

ERROR_SHARING_BUFFER_EXCEEDED

ERROR_SHARING_VIOLATION

Another process has the file open.

ERROR_TOO_MANY_OPEN_FILES

Comments

The **DosOpen** function opens the file whose name matches the name pointed to by the *pszFileName* parameter. The name must be a valid MS OS/2 filename and must not contain wildcard characters. Also, it cannot be the name of a volume or directory. The *fOpenFlags* parameter specifies whether the function is to open an existing file, create a new file, or open and truncate an existing file. The *fOpenMode* parameter specifies whether the file is to be opened for reading, writing, or both. This parameter also specifies the sharing mode; that is, it specifies whether other processes may open the file while the current process has it open.

Every file has associated file attributes, such as read-only or hidden attributes. The **DosOpen** function opens a file only if the access method specified by *fOpenMode* is compatible with the file attributes. For example, **DosOpen** cannot open a read-only file for writing. Also, **DosOpen** cannot open a file if some other process opened it with a sharing method that denies access to other processes. For example, the **DosOpen** function cannot open a file that is already open with the deny-write-access sharing method.

When **DosOpen** creates or truncates a file, it sets the file to the size specified by the *ulFileSize* parameter. If *ulFileSize* is less than the original size, the function truncates the file and any truncated data is lost. If *ulFileSize* is greater than the file's original size, the function appends new bytes to the end of the file. The value of the new bytes is undefined. Whenever the function appends new bytes, it tries to allocate space for the bytes that is contiguous (or nearly contiguous) with the existing file.

When **DosOpen** creates a file, it sets the file attributes to those specified by the *usAttribute* parameter. When **DosOpen** truncates a file, it may also set the file attributes, but only if the specified attributes are compatible with the file's existing attributes. For example, **DosOpen** cannot update the attributes of a file that has the read-only attribute.

After opening or creating a file, **DosOpen** positions the file pointer at the first byte in the file. The process may change this position by using the **DosChgFilePtr** function.

The *fOpenMode* parameter is a combination of one or more of the values given in Table 3.1. Specifically, it consists of one access mode (0x0000, 0x0001, or 0x0002) and one share mode (0x0010, 0x0020, 0x0030, or 0x0040). The other values are optional and may be given in any combination:

Table 3.1

Open Modes for fOpenMode Parameter

Value	Meaning
0x0000	Read-only access mode. Data may be read from the file, but not written to it.
0x0001	Write-only access mode. Data may be written to the file, but not read from it.
0x0002	Read-write access mode. Data may be read from or written to the file.
0x0010	Deny read-write share mode. The current process has exclusive access to the file. The file cannot be opened by any other process (including the current process) until the file is closed.
0x0020	Deny-write share mode. Other processes may open the file for read-only access, but cannot open it for write-only or read-write access until the file is closed.
0x0030	Deny-read share mode. Other processes may open the file for write-only access, but cannot open it for read-only or read-write access until the file is closed.
0x0040	Deny-none share mode. Other processes can open the file for any access: read-only, write-only, or read-write.
0x0080	Inheritance flag. If specified, the file handle is private to the current process; that is, the handle is not automatically available to any child process the current process may start. If not specified, any child process started by the current process inherits the file handle; that is, the child process may use the handle without first creating it by using the DosOpen function. When a file is inherited by a child process, all sharing and access restrictions are also inherited. MS OS/2 does not provide a built-in method to inform a child process that has inherited a given file. The parent process must pass this information to the child process.
0x2000	Fail-on-error flag. If specified, any function that uses the file handle returns immediately with an error code if there is an I/O error, such as the disk-drive door is open or there is a missing sector. If not specified, the system passes the error to the system critical-error handler which reports the error to the user. The fail-on-error flag applies to all functions that use the file handle, with the exception of the DosDevIOCtl function. The DosDevIOCtl function always returns error codes without reporting them to the user, regardless of whether or not the fail-on-error flag is given. If a critical error occurs that the program cannot process, the program must use the DosSetFHandState function to turn off the fail-on-error flag. Then it must call the input or output function that caused the error. This lets the system critical-error handler process the error. The fail-on-error flag is not inherited by child processes.

Table 3.1 (*continued*)

Value	Meaning
0x4000	Write-through flag. This flag applies to functions, such as DosWrite , that use the file handle and write data to the file. If specified, the system writes data to the device before the given function returns. If not specified, the system may store data in an internal file buffer. It then only writes the data to the device when the buffer is full or when it is explicitly flushed by using the DosClose or DosBufReset function. In either case, the system may use the internal file buffer.
0x8000	Direct-access-storage-device (DASD) flag. If specified, the file handle represents a physical drive that has been opened for direct access. The <i>pszFileName</i> parameter must specify a drive name. The file handle can be used to bypass the file system and directly access the sectors of the drive by using the DosDevIOCtl function. File handles with the DASD flag are intended to be used by system programs and not by application programs. If a physical drive is opened for direct access, the LOCKDRIVE function (0x0008, 0x0000) should be used to lock the drive, preventing other processes from attempting to access the drive. The LOCKDRIVE function is described in Chapter 4, "Input-and-Output Control Functions."

When a file is closed, any sharing restrictions placed on it by the opening process are canceled.

There are a number of MS OS/2 functions that retrieve and set file information such as file attributes, dates and times of creation, and access and sharing methods. For example, a file with a read-only attribute can be given the normal file attribute by using the **DosSetFileMode** function.

The file handle created by **DosOpen** can be duplicated by using the **DosDupHandle** function. The duplicated file handle inherits all sharing and access restrictions.

The **DosOpen** function can be used to provide a simple file-existence semaphore. For example, setting the *fOpenFlags* parameter to 0x0010 causes **DosOpen** to fail if the file exists, and to succeed if the file does not exist.

Family API Restrictions

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

- There are restrictions on the values that may be used with the *fOpenMode* parameter. The parameter can be a combination of the following values:

Value	Meaning
0x0000	Read-only access mode.
0x0001	Write-only access mode.
0x0002	Read/write access mode.
0x0010	Deny-read/write share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0020	Deny-write share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0030	Deny-read share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0040	Deny-none share mode. Not available in MS-DOS 2.x. Only available in MS-DOS 3.x when the share command has been used.
0x0080	Inheritance flag. Not available in MS-DOS 2.x.
0x4000	Write-through flag. Not available in MS-DOS 2.x.
0x8000	Direct-access-storage-device (DASD) flag.
The fail-on-errors flag (0x2000) is not available for real-mode programs.	

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 open flag is set to 0x0010 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,       /* address to store action taken */
        100L,           /* size of new file         */
        0,              /* file's attribute         */
        0x10,           /* action to take if file exists */
        0x41,           /* file's open mode         */
        0L);            /* Reserved                  */

```

See Also

DosChgFilePtr, DosDupHandle, DosExecPgm, DosQFHandState, DosQFileInfo, DosQFileMode, DosQFSInfo, DosSetFHandState

- **USHORT DosOpenQueue**(*ppidOwner, phqueue, pszQueueName*)
PUSHORT *ppidOwner*; pointer to variable for queue owner's PID
PHQUEUE *phqueue*; pointer to variable for handle of queue
PSZ *pszQueueName*; pointer to queue-name

The **DosOpenQueue** function opens a queue for the current process.

Parameter	Description
<i>ppidOwner</i>	Points to the variable that receives the process identifier of the queue owner.
<i>phqueue</i>	Points to the variable that receives the handle of the queue.
<i>pszQueueName</i>	Points to a null-terminated string. The string identifies the queue and must have the following form: \queues\name The string name, <i>name</i> , must have the same format as an MS OS/2 filename and must identify a queue that has been previously created by using the Dos-CreateQueue function. For example, the name \queues\abc.ext is acceptable.

Return Value

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

- ERROR_QUE_NAME_NOT_EXIST**
There is no queue with the specified name.
- ERROR_QUE_NO_MEMORY**
The queue segment is full.

See Also

DosCloseQueue, **DosCreateQueue**, **DosReadQueue**,
DosWriteQueue

- **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, clear, and carry out other tasks with the semaphore.

Parameter	Description
<i>phsem</i>	Points to the variable that receives the new semaphore handle.
<i>pszSemName</i>	Points to a null-terminated string. The string identifies the semaphore and must have the following form: \sem\name The string name, <i>name</i> , must have the same format as an MS OS/2 filename and must identify a semaphore that has been previously created by using the DosCreateSem function. For example, the name \sem\abc.ext is acceptable.

Return Value

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

ERROR_INVALID_NAME
The semaphore name is invalid.

ERROR_SEM_NOT_FOUND
The semaphore does not exist.

ERROR_TOO_MANY_SEMAPHORES

Comments

The **DosOpenSem** function returns only the handle of the semaphore; it does not test or change the value of the semaphore. The semaphore handle is equal to 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 on a disk file. When the last process that has 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 previously been created:

```
HSEM hsem;
DosOpenSem(&hsem, "\\SEM\\ABC.EXT"); /* Opens the semaphore */
.
.
DosCloseSem(hsem); /* Closes the semaphore */
```

See Also

DosCloseSem, DosCreateSem, DosSemClear, DosSemRequest

■ **USHORT DosPeekQueue**(*hqueue, pulResult, pusDataLength, pulDataAddr, pusElementCode, fNoWait, pbElemPrty, hsem*)

HQUEUE <i>hqueue</i> ;	handle of queue to read from
PULONG <i>pulResult</i> ;	Requests identification data
PUSHORT <i>pusDataLength</i> ;	length of element received
PULONG <i>pulDataAddr</i> ;	address of element received
PUSHORT <i>pusElementCode</i> ;	indicator of element received
USHORT <i>fNoWait</i> ;	Indicates no wait if queue is empty
PBYTE <i>pbElemPrty</i> ;	priority of element
HSEM <i>hsem</i> ;	semaphore handle

The **DosPeekQueue** function retrieves an element without removing it from a queue. It copies the element to the buffer pointed to by the *pulDataAddr* parameter and fills the structure pointed to by the *pulResult* parameter with information about the element. The **DosPeekQueue** function looks for an element either at the beginning of the queue or immediately after a specified element, depending on the value of the variable pointed to by the *pusElementCode* parameter. When the function finds an element, it copies the element identifier in the same variable. The element identifier can be used to search for the next element or to read the given element from the queue.

If the queue is empty, **DosPeekQueue** either returns immediately or waits for an element to be written to the queue, depending on the value of the *fNoWait* parameter. If the function waits, the function clears the semaphore identified by the *hsem* parameter as soon as the element is retrieved. The semaphore may be either a RAM or a system semaphore.

Parameter	Description
<i>hqueue</i>	Identifies the queue to be searched. The handle must have been previously created or opened by using the DosCreateQueue or DosOpenQueue function.
<i>pulResult</i>	Points to a structure that receives information about the request. For a full description, see the following “Structures” section.
<i>pusDataLength</i>	Points to the variable that receives the actual number of bytes copied to the buffer pointed to by the <i>pulDataAddr</i> parameter.
<i>pulDataAddr</i>	Points to the buffer that receives the element being retrieved from the queue.
<i>pusElementCode</i>	Points to the variable that specifies where to look in the queue for an element. If the <i>pusElementCode</i> parameter is 0x0000, the function looks at the beginning of the queue. Otherwise, the function looks for the element that immediately follows the specified element. In this case, the value is assumed to be an element identifier. When the function returns, it copies the identifier of the retrieved element to the variable, overwriting the previous value.
<i>fNoWait</i>	Specifies whether or not to wait for an element to be placed in the queue. If the <i>fNoWait</i> parameter is DCWW_WAIT , the function waits until an element is available. If it is DCWW_NOWAIT , the function returns immediately.
<i>pbElemPrty</i>	Points to a variable that receives the priority specified when the element was added to the queue. This is a numeric value from 0 to 15; 15 is the highest priority.
<i>hsem</i>	Identifies a semaphore. The 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. The error value may be one of the following:

ERROR_QUE_ELEMENT_NOT_EXIST

The requested element is not present.

ERROR_QUE_EMPTY

The queue is empty.

ERROR_QUE_INVALID_HANDLE

The queue handle is invalid.

ERROR_QUE_INVALID_WAIT

An invalid wait option was specified.

ERROR_QUE_PROC_NOT_OWNED

A process does not own a queue.

Structures

The organization and content of the structure pointed to by the *pulResult* parameter are identical to those of the following structure:

```
struct {
    PID pidProcess;
    USHORT usEventCode;
};
```

Field	Description
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 wish to make of it.

Comments

Only the process that owns the queue may call the **DosPeekQueue** function. The process that created the queue owns it.

The semaphore provided with the function would typically be used in a call to the **DosMuxSemWait** function in order to wait until the queue has an element. If the *fNoWait* parameter is **DCWW_NOWAIT**, the *hsem* parameter is ignored.

See Also

DosCreateQueue, DosMuxSemWait, DosReadQueue

■	USHORT DosPhysicalDisk (<i>usFunction</i> , <i>pbOutBuf</i> , <i>cbOutBuf</i> , <i>pbParmBuf</i> , <i>cbParmBuf</i>)	
	USHORT <i>usFunction</i> ;	type of information
	PBYTE <i>pbOutBuf</i> ;	pointer to output buffer
	USHORT <i>cbOutBuf</i> ;	output-buffer length
	PBYTE <i>pbParmBuf</i> ;	pointer to user-supplied information
	USHORT <i>cbParmBuf</i> ;	length of user-supplied information

The **DosPhysicalDisk** function retrieves information about partitionable disks.

Parameter	Description								
<i>usFunction</i>	Specifies the type of information to retrieve from the disk. It can be one of the following values:								
	<table border="1"> <thead> <tr> <th>Value</th><th>Meaning</th></tr> </thead> <tbody> <tr> <td>0x0001</td><td>Retrieve the total number of partitionable disks.</td></tr> <tr> <td>0x0002</td><td>Retrieve a handle to use with Category 9 IOCTL functions.</td></tr> <tr> <td>0x0003</td><td>Retrieve a handle for a partitionable disk.</td></tr> </tbody> </table>	Value	Meaning	0x0001	Retrieve the total number of partitionable disks.	0x0002	Retrieve a handle to use with Category 9 IOCTL functions.	0x0003	Retrieve a handle for a partitionable disk.
Value	Meaning								
0x0001	Retrieve the total number of partitionable disks.								
0x0002	Retrieve a handle to use with Category 9 IOCTL functions.								
0x0003	Retrieve a handle for a partitionable disk.								
<i>pbOutBuf</i>	Points to the buffer that receives output information. For a full description, see Table 3.2.								
<i>cbOutBuf</i>	Specifies the length in bytes of the output buffer.								
<i>pbParmBuf</i>	Points to a buffer that contains parameter data. For a full description, see Table 3.3.								
<i>cbParmBuf</i>	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

The organization and content of the output buffer depend on the given function. Table 3.2 lists the values used for each function:

Table 3.2

pbOutBuf

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.

The organization and content of the parameter buffer depend on the given function. Table 3.3 lists the values used for each function:

Table 3.3

pbParmBuf

Function	cbParmBuf	Input Parameters
1	0	None. Must be zero.
2	Length of string	A null-terminated string that specifies the partitionable disk. The string must have the following form: <i>number</i> : The <i>number</i> parameter specifies the partitionable disk number. Partitionable disk numbers start at 1.
3	2	Handle retrieved by function 2.

A partitionable disk handle can only be used with the **DosDevIOCtl** function for the Category 9 IOCTL functions. Using the handle for a physical partitionable disk is not permitted in handle-based file system functions, such as **DosRead** or **DosClose**.

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 usOutBuf;  
DosPhysicalDisk(1, /* type of information to obtain */  
    (PBYTE) &usDataBuffer, /* address of data buffer */  
    2, /* length of data buffer */  
    OL, /* pointer to parameter list */  
    0); /* length of parameter list */
```

See Also

DosDevConfig, DosDevIOCtl

- **USHORT DosPortAccess**(*usReserved*, *fFunction*, *usFirstPort*, *usLastPort*)
USHORT *usReserved*; Must be zero
USHORT *fFunction*; request or release
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.

Parameter	Description
<i>usReserved</i>	Specifies a reserved value. It must be zero.
<i>fFunction</i>	Specifies the type of access request. If this parameter is 0x0000, the function requests access to a port. If it is 0x0001, the function releases access to a port.
<i>usFirstPort</i>	Specifies either the starting port number (start-of-range) in a contiguous range or a single port.
<i>usLastPort</i>	Specifies either the ending port number (end-of-range) in a contiguous range or a single port. If only one port is being used, both the <i>usFirstPort</i> and <i>usLastPort</i> parameters must be set to the same port number.

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.

For a program (no IOPL segments) that accesses a device through a device driver or through an interface package such as VIO, the application does not need to gain port access. Instead, the device driver or interface package must obtain the necessary I/O access.

Note that 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

■ **USHORT DosPTrace(*pvoidPtraceBuf*)**
PVOID *pvoidPtraceBuf*; pointer to a structure

The **DosPTrace** function provides access to the MS OS/2 debugging functions. These functions are available to any process that has started a protected-mode child process by using the **DosExecPgm** function with the *fExecFlags* parameter set to EXEC_TRACE.

Parameter	Description
<i>pvoidPtraceBuf</i>	Points to a structure. For a full description, see the following "Structures" section.

Return Value

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

ERROR_ACCESS_DENIED

The specified process identifier is not enabled for tracing.

ERROR_INVALID_FUNCTION

The function requested by **DosPTrace** is invalid.

ERROR_INVALID_PROCID

The specified process identifier does not exist.

Structures

The structure pointed to by the *pvoidPtraceBuf* parameter has the following form:

```
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;
};
```

Field	Description																		
pid	Specifies the process identifier of the program that is being debugged.																		
tid	Specifies the thread identifier of the program that is being debugged.																		
cmd	Specifies the command to carry out. It can be one of the following values:																		
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>Read memory I-space.</td></tr><tr><td>0x0002</td><td>Read memory D-space.</td></tr><tr><td>0x0003</td><td>Read registers.</td></tr><tr><td>0x0004</td><td>Write memory I-space.</td></tr><tr><td>0x0005</td><td>Write memory D-space.</td></tr><tr><td>0x0006</td><td>Write registers.</td></tr><tr><td>0x0007</td><td>Go (with signal).</td></tr><tr><td>0x0008</td><td>Terminate child process.</td></tr></table>	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.
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 to receive the floating-point register values.
0x000F	Set floating-point registers. The segv and offv fields must specify the address of a 94-byte buffer containing 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 to the buffer to receive 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 to the field specifying the result of the command. 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.

	0xFFFF8	Library module has just been loaded. The value field contains the library-module handle.
	0xFFFF7	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		Register AX .
rBX		Register BX .
rCX		Register CX .
rDX		Register DX .
rSI		Register SI .
rDI		Register DI .
rBP		Register BP .
rDS		Register DS .
rES		Register ES .
rIP		Register IP .
rCS		Register CS .
rF		Flags.
rSP		Register SP .
rSS		Register SS .

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, set break points, and read and write to the child-process instruction and data memory. The **DosPTrace** function lets the parent process directly access the child process's memory to insert break points or change data.

The parent process starts the program to be debugged by using the **DosExecPgm** function with the *fExecFlags* parameter set to **EXEC_TRACE**. The parent process stops the child process by using the **DosPTrace** function with the **cmd** field of the structure pointed to by the *pvoidPtraceBuf* parameter 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. On the return from the child process, **DosPTrace** fills the structure pointed to by the *pvoidPtraceBuf* parameter with the current values of the child process's registers and a code that indicates the reason for returning.

The **DosPTrace** function can be used to debug a process with multiple threads. In such a case, the **tid** field in the structure pointed to by the *pvoidPtraceBuf* parameter must be set to the identifier of the thread to be debugged. Other threads in the process are suspended. For a process with multiple threads, the address space is the same for all the threads in the process. Also, issuing the commands to read/write memory locations, or set break points, affects all the threads in the process even though the command was issued with a specific thread identifier. If the parent process uses the 0x000B command, the debugger may have a selective thread or group of threads running while others are suspended. This action is useful in allowing only these selected threads to be affected by the break points, and in manipulating them while the other threads remain suspended.

For information on 0x000E and 0x000F commands and on the format of the buffer (that is, the order of the floating registers), see *The iAPX 286 Programmer's Reference Manual* (1987, Intel Corporation).

See Also

DosExecPgm, **DosGetInfoSeg**

- **USHORT DosPurgeQueue(*hqueue*)**
HQUEUE *hqueue*; handle of queue to be purged

The **DosPurgeQueue** function purges a queue of all elements.

Parameter	Description
<i>hqueue</i>	Identifies the queue to be purged. The handle must have been previously created by using the DosCreateQueue function.

Return Value

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

- ERROR_QUE_INVALID_HANDLE**
The queue handle is invalid.
- ERROR_QUE_PROC_NOT_OWNED**
A process must own a queue to purge it.

Comments

Only the process that owns the queue may call the **DosPurgeQueue** function. The process that created the queue owns it.

See Also

DosCreateQueue

- **USHORT DosPutMessage(*hf*, *cbMsg*, *pchMsg*)**
HFILE *hf*; handle of the 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 *pchMessage* parameter to the file identified by the *hf* parameter.

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

Parameter	Description
<i>hf</i>	Identifies the file that receives the message. The handle must have been previously created by using the DosOpen function. Standard file handles (such as 1 and 2) may also be used.

cbMsg Specifies the length in bytes of the message that is output.

pchMsg Points to the message that is output.

Return Value

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

ERROR_INVALID_HANDLE
The file handle is invalid.

ERROR_MR_UN_PERFORM
Unable to perform the requested function.

ERROR_WRITE_PROTECT
Cannot write to write-protected disk.

Comments

The screen width is assumed to be 80 columns. So if a word is about to span column 80, that word is “wrapped” 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 assumes that the starting position of the cursor is column 1.

Family API 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**

■ **USHORT** **DosQCurDir**(*usDriveNumber*, *pPathBuf*, *pcbPathBuf*)
USHORT *usDriveNumber*; drive number
PBYTE *pPathBuf*; directory-path buffer
PUSHORT *pcbPathBuf*; length of directory-path buffer

The **DosQCurDir** function retrieves the path of the current directory on the specified drive. The function copies a null-terminated string to the

buffer pointed to by the *pPathBuf* parameter. The string identifies the current directory and consists of one or more directory names separated by backslashes (\).

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

Parameter	Description
<i>usDriveNumber</i>	Specifies the drive number. For example, the default drive is 0, drive A is 1, and so on.
<i>pPathBuf</i>	Points to a buffer that receives the path of the current directory.
<i>pcbPathBuf</i>	Points to the variable that receives the length in bytes of the retrieved path.

Return Value

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

ERROR_BUFFER_OVERFLOW

ERROR_DRIVE_LOCKED

ERROR_INVALID_DRIVE

ERROR_NOT_DOS_DISK

Comments

The drive letter is not part of the returned string.

Example

This example calls the **DosQCurDisk** function to get the current drive number, and then calls the **DosQCurDir** function using the current drive number to get the current directory path:

```
CHAR achPath[128];                /* buffer for pathname */
USHORT cbPath = sizeof(achPath), usDisk;
ULONG ulLogicalDriveMap;
DosQCurDisk(&usDisk, &ulLogicalDriveMap); /* Gets current drive */
DosQCurDir(usDisk,                /* drive number */
            achPath,               /* buffer for directory path */
            &cbPath);             /* length of directory buffer */
```


See Also

DosChdir, DosQCurDisk, DosSelectDisk

■ **USHORT DosQCurDisk**(*pusDriveNumber*, *pulLogicalDrives*)
PUSHORT *pusDriveNumber*; default drive number
PULONG *pulLogicalDrives*; drive-map area

The **DosQCurDisk** function retrieves the current drive number and a logical drive map. 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 without affecting other processes.

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

Parameter	Description
<i>pusDriveNumber</i>	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.
<i>pulLogicalDrives</i>	Points to the variable that receives the logical drive map.

Return Value

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

Comments

The logical drive map identifies which of the 26 possible disk drives are available. 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 set to 0, the drive does not exist.

A process can change the current drive by using the **DosChdir** function.

The default current drive for a process is the drive on which the process is called.

Example

This example calls the **DosQCurDisk** function to determine the current default drive and how many logical drives exist. It then uses a **for** loop to display the letter of every logical drive by checking whether its bit in the *ulLogicalDriveMap* variable is set:

```
CHAR chDrives;
USHORT usDisk;
ULONG ulLogicalDriveMap;
DosQCurDisk(&usDisk, &ulLogicalDriveMap); /* Gets current drive */
for (chDrives = 'A'; chDrives <= 'Z'; chDrives++) {
    if (ulLogicalDriveMap & 1) /* If the drive bit is set, */
        VioWrtTty(&chDrives, 1, 0); /* displays the drive letter */
    ulLogicalDriveMap >>= 1;
}
```

See Also

DosChdir, DosQCurDir, DosSelectDisk

■ **USHORT DosQFHandState**(*hf*, *pfsStateFlags*)
HFILE *hf*; file handle
PUSHORT *pfsStateFlags*; 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.

Parameter	Description
<i>hf</i>	Identifies the file whose state is to be retrieved. The handle must have been previously created by using the DosOpen function.
<i>pfsStateFlags</i>	Points to the variable that receives the file-handle state. For a complete list of possible values, see Table 3.4.

Return Value

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

ERROR_INVALID_HANDLE

Comments

The file-handle state consists of one access mode (0x0000, 0x0001, or 0x0002) and one share mode (0x0010, 0x0020, 0x0030, or 0x0040). Other values are optional and may be given in any combination. The file-handle state is a combination of the values shown in Table 3.4:

Table 3.4
File-Handle State

Value	Meaning
0x0000	Read-only access mode. Data may be read from the file, but not written to it.
0x0001	Write-only access mode. Data may be written to the file, but not read from it.
0x0002	Read-write access mode. Data may be read from or written to the file.
0x0010	Deny-read/write share mode. The current process has exclusive access to the file. The file cannot be opened by any other processes (including the current process) until the file is closed.
0x0020	Deny-write share mode. Other processes may open the file for read-only access, but cannot open it for write-only or read-write access until the file is closed.
0x0030	Deny-read share mode. Other processes may open the file for write-only access, but cannot open it for read-only or read-write access until the file is closed.
0x0040	Deny-none share mode. Other processes can open the file for any access: read-only, write-only, or read-write.
0x0080	Inheritance flag. If the value is specified, the file handle is private to the current process; that is, the handle is not automatically available to any child process started by the current process. If it is not specified, any child process started by the current process inherits the file handle; that is, the child process may use the handle without first creating it by using the DosOpen function.
0x2000	Fail-on-error flag. If this value is specified, any function that uses the file handle returns immediately with an error code if there is an input/output error, such as the disk-drive door is open or there is a missing sector. If this value is not specified, the system passes the error to the system critical-error handler which reports the error to the user. The fail-on-error flag applies to functions that use the file handle except the DosDevIOCtl function. The DosDevIOCtl function always returns error codes without reporting them to the user, even if the fail-on-error flag is given. The fail-on-error flag is not inherited by child processes.

Table 3.4 (continued)

Value	Meaning
0x4000	Write-through flag. This flag applies to functions, such as DosWrite , that use the file handle and write data to the file. If specified, the system writes data to the device before the given function returns. If not specified, the system stores data in an internal file buffer and only writes the data to the device when the buffer is full, or when it is explicitly flushed by using the DosClose or DosBufReset function. In either case, the system may use the internal file buffer.
0x8000	Direct-access-storage-device (DASD) flag. If this value is specified, the file handle represents a physical drive that has been opened for direct access. The file handle can be used to bypass the file system and to directly access the sectors of the drive by using the DosDevIOCtl function. File handles with the DASD flag are intended to be used by system programs and not by application programs.

Example

This example calls the **DosQFHandState** function using the handle of a previously opened file, and then checks the *usFileHandleState* variable to report if the file is opened for read/write access:

```
HFILE hf;
USHORT fsStateFlags;
.
.
.
DosQFHandState(hf, &fsStateFlags);
if (fsStateFlags & 0x02)
    VioWrtTTY("File opened for read/write access\n\r", 35, 0);
```

See Also

DosDevIOCtl, **DosExecPgm**, **DosOpen**

■ **USHORT DosQFileInfo(hf, usInfoLevel, pfstsInfo, cbInfoBuf)**
HFILE hf; file handle
USHORT usInfoLevel; 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 defines 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. If any of the returned times are zero, the file system in which the file resides does not support that time.

The file information is based on the most recent call to the **DosClose** or **DosBufReset** function.

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

Parameter	Description
<i>hf</i>	Identifies the file whose information is to be retrieved. The handle must have been previously created by using the DosOpen function.
<i>usInfoLevel</i>	Specifies the level of file information required. It must be 0x0001.
<i>pfstsInfo</i>	Points to the buffer that receives the file information. For a full description, see the following “Structures” section.
<i>cbInfoBuf</i>	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. The error value may be one of the following:

ERROR_BUFFER_OVERFLOW
 ERROR_DIRECT_ACCESS_HANDLE
 ERROR_INVALID_HANDLE
 ERROR_INVALID_LEVEL

Structures

The organization and content of the buffer that receives the file information is identical to the following structure:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

Field	Description
fdateCreation	Specifies date of file creation.
ftimeCreation	Specifies time of file creation.
fdateLastAccess	Specifies date of last file access.
ftimeLastAccess	Specifies time of last file access.
fdateLastWrite	Specifies date of last write to the file.
ftimeLastWrite	Specifies time of last write to the file.
cbFile	Specifies end of file data.
cbFileAlloc	Specifies file size allocated.
attrFile	Specifies attributes.

Example

This example opens a 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, OL, O, Ox11, Ox42, OL);
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

DosClose, DosQFileMode, DosSetFileInfo

- **USHORT DosQFileMode**(*pszFileName, pusAttribute, ulReserved*)
PSZ *pszFileName*; pointer to filename
PUSHORT *pusAttribute*; pointer to variable for file's attribute
ULONG *ulReserved*; Must be zero

The **DosQFileMode** function retrieves the mode of the specified file. The file mode (also called the file attributes) represents additional information about a file that may affect how programs access the file. For example, the read-only attribute prevents programs from opening a file for writing. The file mode is 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.

Parameter	Description														
<i>pszFileName</i>	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.														
<i>pusAttribute</i>	Points to the variable that receives the file mode. It can be a combination of the following values:														
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>Normal file</td></tr> <tr> <td>0x0001</td><td>Read-only file</td></tr> <tr> <td>0x0002</td><td>Hidden file</td></tr> <tr> <td>0x0004</td><td>System file</td></tr> <tr> <td>0x0010</td><td>Subdirectory</td></tr> <tr> <td>0x0020</td><td>Archived file</td></tr> </table>	Value	Meaning	0x0000	Normal file	0x0001	Read-only file	0x0002	Hidden file	0x0004	System file	0x0010	Subdirectory	0x0020	Archived file
Value	Meaning														
0x0000	Normal file														
0x0001	Read-only file														
0x0002	Hidden file														
0x0004	System file														
0x0010	Subdirectory														
0x0020	Archived file														
<i>ulReserved</i>	Specifies a reserved value. It must be zero.														

Return Value

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

ERROR_DRIVE_LOCKED
 ERROR_FILE_NOT_FOUND
 ERROR_NOT_DOS_DISK
 ERROR_PATH_NOT_FOUND

Comments

The **DosQFileMode** function does not recognize a volume label and therefore does not retrieve the file mode for these labels. The **DosQFSInfo** function may be used for this purpose.

Example

This example calls the **DosQFileMode** function and displays a message if the filename *abc* is a subdirectory:

```
USHORT usMode;  
DosQFileMode("abc", /* filename */  
             &usMode, /* address of variable for file's attribute */  
             0L); /* Must be zero */  
if (usMode == 0x10)  
    VioWrtTTY("abc is a subdirectory\n\r", 23, 0);
```

See Also

DosQFHandState, DosQFSInfo, DosSetFileMode

- **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 system information defines characteristics of the disk, such as size and available memory.

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

Parameter	Description
<i>usDriveNumber</i>	Specifies the logical drive number for the device whose information is to be retrieved. It can be any value from 0 to 26. If this parameter is zero, information about the current drive is retrieved. Otherwise, 1 specifies drive A, 2 specifies drive B, and so on.
<i>usInfoLevel</i>	Specifies the level of file information to be retrieved. It may be 1 or 2. For a full description, see the following "Structures" section.
<i>pbInfo</i>	Points to the buffer that receives the file system information. The buffer size depends on the level of file-system information requested. For a full description, see the following "Structures" section.
<i>cbInfo</i>	Specifies the length in bytes of buffer that receives file-system information.

Return Value

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

ERROR_BUFFER_OVERFLOW
 ERROR_INVALID_DRIVE
 ERROR_INVALID_LEVEL
 ERROR_NO_VOLUME_LABEL

Structures

Level 1 file-system information has an organization and content that is identical to the following structure:

```
typedef struct _FSALLOCATE {
    ULONG idFileSystem;
    ULONG cSectorUnit;
    ULONG cUnit;
    ULONG cUnitAvail;
    USHORT cbSector;
} FSALLOCATE;
```

Field	Description
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.

Level 2 file-system information has an organization and content that is identical to the following structure:

```
struct {
    FDATE fdateCreation;
    FTIME ftimeCreation;
    CHAR cchName;
    CHAR achName[14];
};
```

Field	Description
fdateCreation	Specifies the creation date of the volume label.
ftimeCreation	Specifies the creation time of the volume label.
cchName	Specifies the length of the volume label (the null-terminating character is not included).
achName[14]	Specifies the null-terminated volume label.

Comments

There are two levels of file-system information. Level 1 file-system information specifies the number of sectors per allocation unit on the disk, number of allocation units, available allocation units, and bytes per sector. Level 2 file-system information defines the volume label and the date and time the label was created.

Any trailing blanks in the volume label are removed when the label is defined.

Example

This examples calls the **DosQFSInfo** function and displays the volume label of drive C:

```
struct {
    FDATE fdateCreation;
    FTIME ftimeCreation;
    CHAR cchName;
    CHAR achVolumeName[14];
} FSInfoBuf;

.
.
.
DosQFSInfo(3,                /* drive number (c:) */
2,                          /* type of information requested */
(PBYTE) &FSInfoBuf,         /* address of buffer */
sizeof(FSInfoBuf));         /* size of buffer */
VioWrtTTY(FSInfoBuf.achVolumeName, FSInfoBuf.cchName, 0);
```

See Also

DosQFHandState, **DosQFileMode**, **DosQFSInfo**

- **USHORT DosQHandType**(*hf*, *pusHandType*, *pusDeviceAttr*)
HFILE *hf*; file handle
PUSHORT *pusHandType*; 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 or a device.

Parameter	Description
<i>hf</i>	Identifies the file. The handle must have been previously created by using the DosOpen function.
<i>pusHandType</i>	Points to the variable that receives the handle type. It is one of the following values:

	Value	Meaning
	0x0000	File-system file
	0x0001	Device
	0x0002	Pipe
<i>pusDeviceAttr</i>		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. The error value 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 examples calls the **DosQHandType** function to determine if standard output has been redirected to a file:

```
USHORT usHandType, usDeviceAttr;
DosQHandType(1,                                /* file handle */
              &usHandType,                      /* type of handle */
              &usDeviceAttr);                  /* device attribute */
if (usHandType == 1)
    VioWrtTTY("stdout is a device\n\r", 20, 0);
else
    VioWrtTTY("stdout is a file\n\r", 18, 0);
```

■ **USHORT DosQueryQueue**(*hqueue*, *pusElemCount*)
HQUEUE *hqueue*; handle of queue
PUSHORT *pusElemCount*; element count

The **DosQueryQueue** function retrieves a count of the number of elements in the given queue. Any process that has a queue open may call the **DosQueryQueue** function.

Parameter	Description
<i>hqueue</i>	Identifies the queue. The handle must have been previously created or opened by using the DosCreateQueue or DosOpenQueue function.
<i>pusElemCount</i>	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. The error value may be the following:

ERROR_QUE_INVALID_HANDLE

The queue handle is invalid, or the queue is closed.

See Also

DosCreateQueue, **DosOpenQueue**

■ **USHORT DosQVerify(*fVerifyOn*)**
PUSHORT *fVerifyOn*; verification on/off

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.

Parameter	Description
<i>fVerifyOn</i>	Specifies the verification mode for the requesting process. The <i>fVerifyOn</i> parameter is TRUE if the system verifies the data. Otherwise, it is 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\n\r", 23, 0);
else
    VioWrtTty("verify mode is not active\n\r", 27, 0);

```

■ **USHORT DosRead**(*hf*, *pvoidBuf*, *cbBuf*, *pcbBytesRead*)

HFILE *hf*; file handle

PVOID *pvoidBuf*; pointer to input buffer

USHORT *cbBuf*; length of buffer

PUSHORT *pcbBytesRead*; pointer to variable for number of bytes read

The **DosRead** function reads one or more bytes of data from the file identified by the *hf* parameter. The **DosRead** function reads up to the specified number of bytes and copies the data to the buffer pointed to by the *pvoidBuf* parameter. It may read less 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).

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

Parameter	Description
<i>hf</i>	Identifies the file to be read. The handle must have been previously created by using the DosOpen function.
<i>pvoidBuf</i>	Points to the buffer that receives the data.
<i>cbBuf</i>	Specifies the number of bytes to be read from the file.
<i>pcbBytesRead</i>	Points to the variable that receives the number of bytes actually read from the file.

Return Value

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

ERROR_ACCESS_DENIED

The file was opened as write-only.

ERROR_BROKEN_PIPE

ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NOT_DOS_DISK

Comments

The **DosRead** function does not return an error when trying to read past the end of the file.

The **DosRead** function reads bytes starting from the current file pointer position. If desired, the file-pointer position can be changed by using the **DosChgFilePtr** function.

Family API Restrictions

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

- This function uses the **KbdStringIn** function whenever the specified file handle identifies the keyboard device.
- In real mode, **KbdStringIn** reads only the number of characters specified in the call, then beeps to signal the user that no additional characters can be entered. In protected mode the user can enter characters until the keyboard buffer is full.

Example

This example creates a **do** loop to read and display the file *abc*:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesRead, cbBytesWritten;
DosOpen("abc", &hf, &usAction, OL, O, Ox01, Ox42, OL);
do {
    DosRead(hf,          /* handle to file          */
            abBuf,       /* address of buffer  */
            sizeof(abBuf), /* size of buffer      */
            &cbBytesRead); /* address to store number of bytes read */
    DosWrite(1, abBuf, cbBytesRead, &cbBytesWritten);
}
while (cbBytesRead);
```

See Also

DosChgFilePtr, DosOpen, DosReadAsync, DosWrite

■ **USHORT DosReadAsync**(*hf*, *hsemRam*, *pusErrCode*, *pvoidBuf*, *cbBuf*,
pcbBytesRead)

HFILE *hf*; file handle
PULONG *hsemRam*; pointer to RAM semaphore
PUSHORT *pusErrCode*; pointer to variable for error return code
PVOID *pvoidBuf*; 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 process continues with other executing.

Parameter	Description
<i>hf</i>	Identifies the file to be read. The handle must have been previously opened by using the DosOpen function.
<i>hsemRam</i>	Points to the RAM semaphore that indicates when the function has finished reading the data.
<i>pusErrCode</i>	Points to the variable that receives any error code the function may generate.
<i>pvoidBuf</i>	Points to the buffer that receives the data to be read.
<i>cbBuf</i>	Specifies the number of bytes to be read from the specified file.
<i>pcbBytesRead</i>	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. The error value may be one of the following:

ERROR_ACCESS_DENIED
The file was opened as write-only.

ERROR_BROKEN_PIPE

ERROR_INVALID_HANDLE

ERROR_LOCK_VIOLATION

ERROR_NO_PROC_SLOTS
Cannot create a thread for this operation.

ERROR_NOT_DOS_DISK

Comments

The **DosReadAsync** function reads up to the specified number of bytes, but it may read less 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).

When **DosReadAsync** has read the data, it clears the RAM semaphore pointed to by the *hsemRam* parameter. If the process intends to use the semaphore to determine when data is available, it must set the semaphore by using the **DosSemSet** function before calling **DosReadAsync**.

If **DosReadAsync** encounters errors while reading data, it copies the error code to the variable pointed to by the *pusErrCode* parameter. The possible error codes are identical to those returned by the **DosRead** function.

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 has occurred.

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, cbBytesWritten;
USHORT usReadReturn;
DosOpen("abc", &hf, &usAction, OL, 0, 0x01, 0x42, OL);
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 */
. /* Does other processing */
DosSemWait(&hReadSemaphore, -1L);
```


See Also

DosOpen, DosRead, DosSemSet, DosSemWait, DosWriteAsync

■ **USHORT DosReadQueue**(*hqueue, pulPidReq, pcbBuf, pulBuf, usElement, fNoWait, pbElemPrty, hsem*)

HQUEUE <i>hqueue</i> ;	handle of queue to read
PULONG <i>pulPidReq</i> ;	pointer to variable for PID and request code
PUSHORT <i>pcbBuf</i> ;	pointer to variable for length of element
PULONG <i>pulBuf</i> ;	pointer to buffer for element
USHORT <i>usElement</i> ;	element number to read
USHORT <i>fNoWait</i> ;	Does not wait if queue is empty
PBYTE <i>pbElemPrty</i> ;	priority of element
HSEM <i>hsem</i> ;	semaphore handle

The **DosReadQueue** function retrieves an element from a queue. It copies the element to the buffer pointed to by the *pulBuf* parameter, then removes the element from the queue and fills the structure pointed to by the *pulPidReq* parameter with information about the element.

Parameter	Description
<i>hqueue</i>	Identifies the queue to be read. The handle must have been previously created or opened by using the DosCreateQueue or DosOpenQueue function.
<i>pulPidReq</i>	Points to a structure that receives information about the request. For a full description, see the following “Structures” section.
<i>pcbBuf</i>	Points to the variable that receives the number of bytes copied to the buffer pointed to by the <i>pulBuf</i> parameter.
<i>pulBuf</i>	Points to the buffer that receives the element being retrieved from the queue.
<i>usElement</i>	Specifies where to look in the queue for an element. If the <i>usElement</i> parameter is 0x0000, the function looks at the beginning of the queue. Otherwise, the function looks for the specified element. The value is assumed to be an element identifier retrieved by using the DosPeekQueue function.
<i>fNoWait</i>	Specifies whether to wait for an element to be placed in the queue. If the <i>fNoWait</i> parameter is DCWW_WAIT, the function waits until an element is available. If it is DCWW_NOWAIT, the function returns immediately.
<i>pbElemPrty</i>	Points to a variable that receives the priority specified when the element was added to the queue. It is a numeric value from 0 to 15; 15 is the highest priority.

hsem Identifies a semaphore. The handle must have been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it must be a RAM semaphore.

Return Value

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

- ERROR_QUE_ELEMENT_NOT_EXIST**
The requested element is not present.
- ERROR_QUE_EMPTY**
Nothing was read; the queue is empty.
- ERROR_QUE_INVALID_HANDLE**
The queue handle is invalid.
- ERROR_QUE_INVALID_WAIT**
An invalid wait option was specified.
- ERROR_QUE_PROC_NOT_OWNED**
A process must own a queue to read it.

Structures

The organization and content of the structure pointed to by the *pulPidReq* parameter is identical to the following structure:

```
struct {  
    PID pidProcess;  
    USHORT usEventCode;  
};
```

Field	Description
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 wish to make of it.

Comments

The **DosReadQueue** function looks for an element at either the beginning of the queue or immediately after a specified element, depending on the value of the variable pointed to by the *usElement* parameter. The beginning of the queue depends on the queue priority. For example, the beginning of a queue with last-in, first-out (LIFO) priority is the last element in the queue.

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 *fNoWait* parameter. If the function waits, it clears the semaphore identified by the *hsem* parameter as soon as the element is retrieved. The semaphore may be either a RAM or system semaphore.

Only the process that owns the queue may call the **DosReadQueue** function. The process that created the queue owns it.

The semaphore provided with the function would typically be used in a call to the **DosMuxSemWait** function to wait until the queue has an element. If the *fNoWait* parameter is `DCWW_WAIT`, *hsem* is ignored.

If **DosReadQueue** reads an empty queue, the requesting thread is placed in a wait state, waiting for an element to be added to the queue. If *fNoWait* is `DCWW_NOWAIT`, the thread does not wait; it returns with a code that indicates there are no entries in the queue.

See Also

DosCreateQueue, DosMuxSemWait, DosPeekQueue, DosWriteQueue

■ **USHORT DosReallocHuge**(*usNumSeg*, *usPartialSize*, *sel*)
USHORT *usNumSeg*; number of 65,536-byte segments
USHORT *usPartialSize*; number of bytes in last segment
SEL *sel*; selector

The **DosReallocHuge** function reallocates the huge memory block specified by the *sel* parameter. 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 that has the number of bytes specified by the *usPartialSize* parameter. If *usPartialSize* is zero, no additional segment is allocated.

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

Parameter	Description
<i>usNumSeg</i>	Specifies the number of 65,536-byte segments allocated.
<i>usPartialSize</i>	Specifies the number of bytes in the last segment. If this parameter is zero, no last segment is allocated.
<i>sel</i>	Specifies the selector for the huge memory block to be reallocated. The selector must have been previously created by using the DosAllocHuge function.

Return Value

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

ERROR_INVALID_PARAMETER

ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosReallocHuge** function does not change the share attribute of the huge memory block. If it was originally a shared block, it remains a shared block. Also, the memory block cannot be reallocated for a size larger than the maximum memory size specified by the *MaxNumSeg* parameter in the original call to the **DosAllocHuge** function.

Each segment in the huge memory block has a unique selector. The selectors are consecutive, with the *sel* parameter specifying the value of the first. The remaining selectors can be computed by adding the selector offset to the first selector one or more times; that is, it is added once for the second selector, twice for the third, and so on. The selector offset is a multiple of two, 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 (that is, $1 \ll 2$ is equal to 4). If the selector offset is 4 and the first selector is 6, the second selector is 10, the third is 14, and so on.

Family API Restrictions

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

- The *usPartialSize* parameter is rounded up to the next paragraph (16-byte) value.

See Also

DosAllocHuge, **DosFreeSeg**, **DosGetHugeShift**

■ **USHORT** **DosReallocSeg**(*usNewSize*, *sel*)
USHORT *usNewSize*; new segment size
SEL *sel*; selector

The **DosReallocSeg** function reallocates the segment specified by the *sel* parameter. 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.

Parameter	Description
<i>usNewSize</i>	Specifies the new size in bytes. It can be any number from 0 to 65,535. If it is zero, the function allocates 65,536 bytes.
<i>sel</i>	Specifies the selector of the segment to be reallocated. The selector must have been previously created by using the DosAllocSeg function.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosReallocSeg** function does not change the shared or discardable attribute of the segment. If it was originally a shared or discardable segment, it remains a shared or discardable segment. The **DosReallocSeg** function cannot reallocate a shared segment to a size smaller than its original size.

If **DosReallocSeg** reallocates a discardable segment, it also locks the segment.

Family API 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, **DosLockSeg**, **DosReallocHuge**

■ **USHORT DosResumeThread(*tid*)**
TID *tid*; thread identifier of thread to be resumed

The **DosResumeThread** function restarts a thread previously stopped by the **DosSuspendThread** function.

Parameter	Description
<i>tid</i>	Specifies the thread identifier of the thread to be resumed. It must have been previously created by using the DosCreateThread function.

Return Value

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

ERROR_INVALID_THREADID

See Also

DosCreateThread, **DosSuspendThread**

- **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.

Parameter	Description
<i>pszDirName</i>	Points to a null-terminated string. The string specifies the directory to be removed and must be a valid MS OS/2 directory name.
<i>ulReserved</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_ACCESS_DENIED
An attempt was made to remove the root directory, or the subdirectory is not empty.

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 or 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:

```
DosDelete("abc\\*.*", OL);           /* Removes all files */
if (DosRmdir("abc", OL))             /* Removes subdirectory */
    VioWrTty("Can't delete subdirectory\n\r", 27, 0);
else {
```

See Also

DosChdir, **DosDelete**, **DosMkdir**

■ **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 segment for a specific 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 provide a way for the user to pass information to a program, for example, a list of directories that might 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 variable's value to the variable pointed to by the *ppszResult* parameter. The first character of the value is the first character following the equal sign (=).

Parameter	Description
<i>pszVarName</i>	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.
<i>ppszResult</i>	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

■ **USHORT DosSearchPath**(*fSearch, pszPath, pszFileName, pbBuf, cbBuf*)

USHORT <i>fSearch</i> ;	search flags
PSZ <i>pszPath</i> ;	pointer to search path or environment variable
PSZ <i>pszFileName</i> ;	pointer to filename
PBYTE <i>pbBuf</i> ;	pointer to result buffer
USHORT <i>cbBuf</i> ;	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 path separated by semicolons (;). The function searches for the filename by looking in each directory (one directory at a time) in the order given.

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 an actual search path (as specified by the *fSearch* parameter). The filename must be a valid MS OS/2 filename, and it may contain wildcard characters if desired. 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 original filename contained wildcard characters, the resulting pathname will also contain

the wildcard characters. It may be necessary to use the **DosFindFirst** function to retrieve the actual filename (or filenames) found in the search path.

Parameter	Description						
<i>fSearch</i>	Specifies how to interpret the <i>pszPath</i> parameter and whether or not to search the current directory. It can be a combination of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>The function searches the current directory before it searches any other directory in the search path. If 0x0001 is not given, the function searches the current directory only if it is explicitly given in the search path.</td></tr> <tr> <td>0x0002</td><td>The <i>pszPath</i> 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 0x0002 is not given, <i>pszPath</i> points to a string that specifies the search path.</td></tr> </table>	Value	Meaning	0x0001	The function searches the current directory before it searches any other directory in the search path. If 0x0001 is not given, the function searches the current directory only if it is explicitly given in the search path.	0x0002	The <i>pszPath</i> 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 0x0002 is not given, <i>pszPath</i> points to a string that specifies the search path.
Value	Meaning						
0x0001	The function searches the current directory before it searches any other directory in the search path. If 0x0001 is not given, the function searches the current directory only if it is explicitly given in the search path.						
0x0002	The <i>pszPath</i> 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 0x0002 is not given, <i>pszPath</i> points to a string that specifies the search path.						
<i>pszPath</i>	Points to a null-terminated string that specifies the search-path reference.						
<i>pszFileName</i>	Points to a null-terminated string that specifies the name of the file to search for. The string must be a valid MS OS/2 filename and may contain wildcard characters.						
<i>pbBuf</i>	Points to the buffer that receives the full pathname of the file if the file is found.						
<i>cbBuf</i>	Specifies the length in bytes of the structure that is pointed to by the <i>pbBuf</i> parameter.						

Return Value

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

Comments

The **DosSearchPath** function does not check for valid 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* file:

```
CHAR achBuf[128];
DosSearchPath(0x0002, /* Uses environment variable */
  "DPATH",           /* Uses DPATH search path */
  "abc.txt",         /* filename */
  achBuf,            /* Receives resulting filename */
  sizeof(achBuf);    /* length of result buffer */
```

The following example is identical to the first example if the DPATH variable is defined as the following:

```
DPATH=c:\sysdir;c:\init
```

```
DosSearchPath(0x0000, /* Uses search path */
  "c:\sysdir;c:\init", /* search path */
  "abc.txt",
  achBuf,
  sizeof(achBuf);
```

See Also

DosFindFirst, DosScanEnv

- **USHORT DosSelectDisk(*usDriveNumber*)**
USHORT *usDriveNumber*; default-drive number

The **DosSelectDisk** function selects the drive that is specified as the default drive for the calling process.

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

Parameter	Description
<i>usDriveNumber</i>	Specifies the number of the default drive. It can be 0x0001 for drive A, 0x0002 for drive B, and so on.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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("\\", OL);         /* Changes to the root directory */
DosOpen("abc.txt", &hf, &usAction, OL, O, 0x11, 0x42, OL);
```

See Also

DosChdir, **DosQCurDisk**

- **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 a parent session may call **DosSelectSession** to switch a session. Also, the parent session, or one of its descendant sessions, must be currently executing in the foreground. The parent session may switch to a child session or it may switch to itself.

Parameter	Description
<i>idSession</i>	Specifies the identifier of the session to be switched to the foreground. This value must have been previously created by using the DosStartSession function. If 0x0000 is given, the function switches the parent session to the foreground.
<i>ulReserved</i>	Specifies a reserved value. It 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 may only be used to select a child session that was created by using the **DosStartSession** function with the **Related** field of the **STARTDATA** structure set to 0x0001. In other words, sessions started as independent sessions may not be selected

through this function. Also, **DosSelectSession** may be called only by the process which originally started the session specified by the *idSession* parameter (through **DosStartSession**).

See Also

DosSetSession, DosStartSession, DosStopSession

■ **USHORT DosSemClear(*hsem*)**
HSEM *hsem*; semaphore handle

The **DosSemClear** function clears a system or RAM semaphore. This function clears a semaphore that has been set by the **DosSemSet** or **DosSemSetWait** function. It releases ownership of a semaphore that has been assigned by the **DosSemRequest** function.

Parameter	Description
<i>hsem</i>	Identifies the semaphore. The handle must have been previously created or opened by using the DosCreateSem or DosOpenSem function, or it must point to a RAM semaphore.

Return Value

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

ERROR_EXCL_SEM_ALREADY_OWNED

ERROR_INVALID_HANDLE

Comments

The **DosSemClear** function cannot clear a system semaphore owned by another process, unless the semaphore is non-exclusive. Non-exclusive semaphores are created by using the **DosCreateSem** function.

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**

■ **USHORT DosSemRequest**(*hsem*, *lTimeOut*)
HSEM *hsem*; semaphore handle
LONG *lTimeOut*; time-out

The **DosSemRequest** function requests that the specified semaphore be set and then waits, if necessary, for the semaphore to be cleared before setting it. If no previous thread has set the semaphore, **DosSemRequest** sets the semaphore and returns immediately. If the semaphore is already set, the function waits until a thread clears the semaphore (using the **DosSemClear** function) or until a time-out occurs.

If the semaphore is set, **DosSemRequest** uses the *lTimeOut* parameter to determine how long to wait. The function can continue to wait until the semaphore is cleared, it can wait for a specified number of milliseconds, or it can return immediately. 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.

Parameter	Description
<i>hsem</i>	Identifies the semaphore. The handle must have been previously created or opened by using the DosCreateSem or DosOpenSem function, or it must point to a RAM semaphore.
<i>lTimeOut</i>	Specifies how long to wait if the semaphore is owned by another process or thread. If the <i>lTimeOut</i> parameter is zero, the function returns immediately if the semaphore is owned. If it is greater than zero, the function waits that number of milliseconds before returning. If it is less than zero, the function continues to wait until the semaphore is no longer owned.

Return Value

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

ERROR_INTERRUPT

ERROR_INVALID_HANDLE

ERROR_SEM_OWNER_DIED

ERROR_SEM_TIMEOUT

ERROR_TOO_MANY_SEM_REQUESTS

Comments

Setting a semaphore with **DosSemRequest** is accumulative. If multiple calls to **DosSemRequest** set the semaphore, multiple calls to the **DosSemClear** function are required to clear the semaphore.

The **DosSemRequest** function cannot set a system semaphore that is set by another process, unless the semaphore is non-exclusive. Non-exclusive semaphores are created by using the **DosCreateSem** function.

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 forever */
DosSemSet(&hsem);             /* Sets the semaphore */
DosSemClear(&hsem);           /* Clears the semaphore */
```

See Also

DosCreateSem, DosExitList, DosOpenSem, DosSemClear,
DosSemSet, DosSemSetWait, DosSemWait

- **USHORT DosSemSet(*hsem*)**
HSEM *hsem*; semaphore handle

The **DosSemSet** function sets the specified semaphore. This function is typically used by a process to set a semaphore, then the process waits for the semaphore to clear by using the **DosSemWait** or **DosMuxSemWait** function.

Parameter	Description
<i>hsem</i>	Identifies the semaphore. The handle must have been previously created or opened by using the DosCreateSem or DosOpenSem function, or it must contain the address of a RAM semaphore.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 non-exclusive. Non-exclusive semaphores are created by using the **DosCreateSem** function.

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, DosSemSet, DosSemWait

- **USHORT DosSemSetWait**(*hsem*, *lTimeOut*)
HSEM *hsem*; semaphore handle
LONG *lTimeOut*; time-out

The **DosSemSetWait** function sets the specified semaphore, and then waits for the semaphore to be cleared. The function waits until a thread clears the semaphore (using the **DosSemClear** function) or until a time-out occurs.

The **DosSemSetWait** function uses the *lTimeOut* parameter to determine how long to wait. The function can continue to wait until the semaphore is cleared, or it can wait for a specified number of milliseconds. 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.

Parameter	Description
<i>hsem</i>	Identifies the semaphore. The handle must have been previously created or opened by using the DosCreateSem or DosOpenSem function, or it must contain the address of a RAM semaphore.
<i>lTimeOut</i>	Specifies how long to wait for the semaphore to be cleared. If it is greater than zero, the function waits no more than that number of milliseconds before returning. If it is less than zero, the function continues to wait until the semaphore is cleared.

Return Value

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

The **DosSemSetWait** function cannot be used to wait for a system semaphore that is owned by another process, unless the semaphore is non-exclusive. Non-exclusive semaphores are created by using the **DosCreateSem** function.

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 handle is retrieved:

```
#include <bseerr.h>

ULONG hsem = 0;
if (DosSemSetWait(&hsem, 5000L)
    == ERROR_SEM_TIMEOUT) {
    . /* error processing */
    .
}
else {
```

See Also

DosCreateSem, **DosOpenSem**, **DosSemClear**, **DosSemWait**

■ **USHORT DosSemWait(*hsem*, *lTimeOut*)**
HSEM *hsem*; semaphore handle
LONG *lTimeOut*; time-out

The **DosSemWait** function waits, if necessary, for the specified semaphore to be cleared. If no previous thread has set the semaphore, **DosSemWait** returns immediately. If the semaphore is already set, the function waits until a thread clears the semaphore (using the **DosSemClear** function) or until a time-out occurs.

If the semaphore is set, **DosSemWait** uses the *lTimeOut* parameter to determine how long to wait. The function can continue to wait until the semaphore is cleared, it can wait for a specified number of milliseconds, or it can return immediately. 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 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.

Parameter	Description
<i>hsem</i>	Identifies the semaphore. The handle must have been previously created or opened by using the DosCreateSem or DosOpenSem function, or it must contain the address of a RAM semaphore.
<i>lTimeOut</i>	Specifies how long to wait if the semaphore is set. If it is zero, the function returns immediately if the semaphore is set. If it is greater than zero, the function waits no more than that number of milliseconds before returning. If it is less than zero, the function continues to wait until the semaphore is cleared.

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 non-exclusive. Non-exclusive semaphores are created by using the **DosCreateSem** function.

Example

This example calls the **DosSemWait** function to get a RAM semaphore. It waits for up to 5 seconds, and then returns an **ERROR_SEM_TIMEOUT** error value if a time-out occurs before the handle is retrieved:

```
ULONG hsem = 0;
if (
    DosSemWait(&hsem, 5000L)
    == ERROR_SEM_TIMEOUT) {
    .
    . /* error processing */
    .
}
else {
```

See Also

DosCreateSem, DosOpenSem, DosSemRequest

■ **USHORT DosSendSignal(*idProcess*, *usSigNumber*)**
USHORT *idProcess*; process identifier of subtree root
USHORT *usSigNumber*; signal to send

The **DosSendSignal** function sends a CONTROL+C or CONTROL+BREAK signal to the last descendant process in a command subtree that has a corresponding signal handler installed.

Parameter	Description
<i>idProcess</i>	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.
<i>usSigNumber</i>	Specifies the signal to send. If this parameter is 0x0001, the CONTROL+C (signal interrupt, SIGINTR) signal is sent. If it is 0x0004, the CONTROL+BREAK (signal break, SIGBREAK) signal is sent.

Return Value

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

Comments

An environment manager may have several applications running at once within a screen group. Each of these applications may have its own family of descendants. MS OS/2 cannot send the termination signal (CONTROL+C or CONTROL+BREAK) to the appropriate process because MS OS/2 does not know which subtree had the keyboard focus when the termination signal was sent.

Instead, the environment manager learns of the termination signal by means of monitors or through keyboard reads in raw mode. Then the environment manager descends the process tree of the process that owns the keyboard focus, looking for the outermost child process. Starting with that process, **DosSendSignal** looks for a descendant process with an installed signal handler. When **DosSendSignal** finds the signal handler, it sends the signal that is specified by the *usSigNumber* parameter. If no process in the subtree has an installed signal handler, **DosSendSignal** returns a unique error value.

DosSendSignal

See Also

DosFlagProcess, DosHoldSignal, DosSetSigHandler

- **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. The default code page is determined by the **codepage** command in the *config.sys* file.

Parameter	Description
<i>usCodePage</i>	Specifies the code-page identifier. It can be any of the possible code-page values described in Appendix C, "Code Pages." If it is 0x0000, the default code page for the system is used.
<i>usReserved</i>	Specifies a reserved value. It 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.

See Also

DosGetCp

- **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.

Parameter	Description
<i>pdateTime</i>	Points to the DATETIME structure that contains the date and time information. For a full description, see the following “Structures” section.

Return Value

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

ERROR_TS_DATETIME
The date or time is invalid.

Structures

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

Field	Description
hours	Specifies the current hour with values from 0 to 23.
minutes	Specifies the current minute with values from 0 to 59.
seconds	Specifies the current second with values from 0 to 59.
hundredths	Specifies the current hundredths of a second with values from 0 to 99.
day	Specifies the current day of the month with values from 1 to 31.
month	Specifies the current month of the year with values from 1 to 12.
year	Specifies the current year with values from 80 to 79 (1980 to 2079).
timezone	Specifies the difference in minutes between the current time zone and Greenwich Mean Time (GMT). This field is positive for time zones to the west of Greenwich, and negative for time zones to the east of Greenwich. For

Eastern Standard Time, this field is 300—five hours after GMT.

weekday Specifies the current day of the week with values from 0 to 6 (Sunday equals zero).

Example

This example retrieves the current date and time. It calls the **DosSetDateTime** function to change the month to September and the day to the 26th:

```
DATEIME 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

■ **USHORT DosSetFHandleState**(*hf*, *fState*)
HFILE *hf*; file handle
USHORT *fState*; file-state flags

The **DosSetFHandleState** function modifies the state of the handle flags for the specified file handle. The function can modify the inheritance, fail-on-error, and write-through flags for any file handle.

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

Parameter	Description
<i>hf</i>	Identifies the handle of the file to be set. It must have been previously created by using the DosOpen function.
<i>fState</i>	Specifies the file-handle state. It can be one or more of the following values:
Value	Meaning
0x0080	Inheritance flag. If this value is specified, the file handle is private to the current process; that is, the handle is not automatically available to any child process that the current process starts. If this value is not specified, any child process started by the

current process inherits the file handle; that is, the child process may use the handle without first creating it by using the **DosOpen** function.

0x2000	Fail-on-error flag. If this value is specified, any function that uses the file handle returns immediately with an error value if there is an input/output error such as 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. For more information, see the DosOpen function.
0x4000	Write-through flag. This flag applies to functions, such as DosWrite , that use the file handle and write data to the file. If this value is specified, the system writes data to the device before the given function returns. If this value is not specified, the system may store data in an internal file buffer. It may then write the data to the device (using the DosClose or DosBufReset function) only when the buffer is full or explicitly flushed. In either case, the system may use the internal file buffer.

Return Value

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

ERROR_INVALID_HANDLE

ERROR_INVALID_PARAMETER

Comments

Setting the write-through flag does not affect any previous writing that may have been done. That data remains in the buffers.

Family API Restrictions

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

- The fail-on-errors and write-through flags (0x2000 and 0x4000) must not be set.

- The inheritance flag (0x0080) must not be set in MS-DOS 2.x.

Example

This example opens the file *abc* with the inheritance flag set to zero (all child processes inherit the file). It gets 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. By setting the inheritance flag, you ensure that any child process that is created will not inherit the file:

```
HFILE hf;
USHORT usAction, fState;
DosOpen("abc", &hf, &usAction, OL, O, 0x01, 0x42, OL);
DosQFHandState(hf, &fState);
DosSetFHandState(hf,                                /* handle to the file */
                  (fState & 0x7F88)                  /* Clears required bits */
                  | 0x80);                            /* Cannot access file */
```

See Also

DosExecPgm, DosOpen, DosQFHandState, DosSetMode

■ **USHORT DosSetFileInfo**(*hf, usInfoLevel, pfstsBuf, cbBuf*)

HFILE <i>hf</i> ;	file handle
USHORT <i>usInfoLevel</i> ;	level of file information
PFILESTATUS <i>pfstsBuf</i> ;	pointer to file-status information
USHORT <i>cbBuf</i> ;	length file-information buffer

The **DosSetFileInfo** function sets the time and date information for the specified file to a new time and date. The function replaces the time and date information of a file with the information given in the structure pointed to by the *pfstsBuf* parameter. The **DosSetFileInfo** function does not work for read-only files.

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

Parameter	Description
<i>hf</i>	Identifies the file whose status information is being changed. The handle must have been previously created by using the DosOpen function.
<i>usInfoLevel</i>	Specifies the level of file information being defined. The <i>usInfoLevel</i> parameter must be 0x0001.
<i>pfstsBuf</i>	Points to the FILESTATUS structure that contains the new file-status information. For a full description, see the following "Structures" section.

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. The error value 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

Structures

The **FILESTATUS** structure pointed to by the *pfstsBuf* parameter 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;
```

Field	Description
fdateCreation	Specifies the date of file creation.
ftimeCreation	Specifies the time of file creation.
fdateLastAccess	Specifies the date of last file access.
ftimeLastAccess	Specifies the time of last file access.
fdateLastWrite	Specifies the date of the last write to the file.
ftimeLastWrite	Specifies the time of the last write to the file.
cbFile	This field is not used by DosSetFileInfo .
cbFileAlloc	This field is not used by DosSetFileInfo .
attrFile	This field is not used by DosSetFileInfo .

Comments

If a field is set to zero, **DosSetFileInfo** does not change the file. For example, if the **ftimeLastAccess** field of the **FILESTATUS** structure is set to zero, the function leaves these values for the file unchanged.

See Also

DosNewSize, **DosQFileInfo**, **DosSetFileMode**

- **USHORT DosSetFileMode(*pszFileName*, *usAttribute*, *ulReserved*)**
PSZ *pszFileName*; filename
USHORT *usAttribute*; new attribute of file
ULONG *ulReserved*; Must be zero

The **DosSetFileMode** function sets the file attributes, or mode, of the specified file.

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

Parameter	Description												
<i>pszFileName</i>	Points to a null-terminated string that specifies the name of the file. The string must be a valid MS OS/2 filename.												
<i>usAttribute</i>	Specifies the file's new attribute. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Normal file</td></tr><tr><td>0x0001</td><td>Read-only file</td></tr><tr><td>0x0002</td><td>Hidden file</td></tr><tr><td>0x0004</td><td>System file (excluded from normal directory searches)</td></tr><tr><td>0x0020</td><td>Archived file</td></tr></table> Some combinations are not allowed; for example, normal file and read-only file.	Value	Meaning	0x0000	Normal file	0x0001	Read-only file	0x0002	Hidden file	0x0004	System file (excluded from normal directory searches)	0x0020	Archived file
Value	Meaning												
0x0000	Normal file												
0x0001	Read-only file												
0x0002	Hidden file												
0x0004	System file (excluded from normal directory searches)												
0x0020	Archived file												
<i>ulReserved</i>	Specifies a reserved value. It must be zero.												

Return Value

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

ERROR_ACCESS_DENIED

Invalid attribute or attempt to set label or subdirectory bits.

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

■ **USHORT DosSetFSInfo**(*usDriveNumber*, *usInfoLevel*, *pbBuf*, *cbBuf*)
USHORT *usDriveNumber*; drive number
USHORT *usInfoLevel*; level of file-system information
PBYTE *pbBuf*; pointer to file-system information structure
USHORT *cbBuf*; length of file-system information buffer

The **DosSetFSInfo** function sets information for a file-system device.

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

Parameter	Description
<i>usDriveNumber</i>	Specifies the logical drive number. The <i>usDriveNumber</i> parameter can be any value from 0 to 26. If it is zero, the default drive is set. Otherwise, 1 specifies drive A, 2 specifies drive B, and so on.
<i>usInfoLevel</i>	Specifies the level of file information required. This parameter must be 0x0002.
<i>pbBuf</i>	Points to a structure that contains the volume-label information. For a full description, see the following "Structures" section.
<i>cbBuf</i>	Specifies the length in bytes of the structure pointed to by the <i>pbBuf</i> parameter.

DosSetFSInfo

Return Value

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

Structures

The structure pointed to by the *pbBuf* parameter has the following form:

```
struct {  
    UCHAR valid_len;  
    char valid[13];  
};
```

Field	Description
valid_len	Specifies the length of the valid field (the null-terminator is not included).
valid[13]	Specifies a null-terminated string that specifies the volume label.

Comments

File-system information can be set only if the volume is opened in a mode that allows write access.

Trailing blanks supplied at the time the volume label is defined are not returned by **DosQFSInfo**.

See Also

DosQCurDisk, **DosQFSInfo**

- **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 more for the default system input/output handles.

Parameter	Description
<i>usHandles</i>	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. The error value may be one of the following:

ERROR_INVALID_PARAMETER

The number of file handles can only be increased, not decreased, or the number of file handles exceeds system limits.

ERROR_NOT_ENOUGH_MEMORY

There is not enough memory.

Comments

This function preserves all currently open file handles.

See Also

DosDupHandle, DosOpen

- **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 execution priority of the specified process or thread. The execution priority affects how the system scheduler distributes execution control to the currently running processes.

Every process and thread belongs to a priority class and has a priority level. The priority classes are time-critical, regular, and idle-time. Time-critical processes and threads are always executed first, and regular processes and threads are executed before idle-time processes and threads. Within a class the priority level defines which process or threads should be executed first. The priority level is a value from 0 to 31.

DosSetPrty changes the priority of a process or thread by changing the priority class and adjusting the priority level.

Within each class, a thread's priority level may vary—either through system action or through **DosSetPrty**. The system changes priority levels through a combination of a specific thread's actions and the overall system activity. The base priority level can be changed within the range 0 to 31.

Parameter	Description										
<i>fScope</i>	Specifies the scope of the request. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>PRTYS_PROCESS</td><td>Priority for the process and all its threads.</td></tr><tr><td>PRTYS_PROCESSTREE</td><td>Priority for the process and all descendant processes.</td></tr><tr><td>PRTYS_THREAD</td><td>Priority for only the thread calling the function.</td></tr></table>	Value	Meaning	PRTYS_PROCESS	Priority for the process and all its threads.	PRTYS_PROCESSTREE	Priority for the process and all descendant processes.	PRTYS_THREAD	Priority for only the thread calling the function.		
Value	Meaning										
PRTYS_PROCESS	Priority for the process and all its threads.										
PRTYS_PROCESSTREE	Priority for the process and all descendant processes.										
PRTYS_THREAD	Priority for only the thread calling the function.										
<i>fPrtyClass</i>	Specifies the priority class of a process or thread. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>PRTYC_NOCHANGE</td><td>No change; leave as is.</td></tr><tr><td>PRTYC_IDLETIME</td><td>Idle-time.</td></tr><tr><td>PRTYC_REGULAR</td><td>Regular.</td></tr><tr><td>PRTYC_TIMECRITICAL</td><td>Time-critical.</td></tr></table>	Value	Meaning	PRTYC_NOCHANGE	No change; leave as is.	PRTYC_IDLETIME	Idle-time.	PRTYC_REGULAR	Regular.	PRTYC_TIMECRITICAL	Time-critical.
Value	Meaning										
PRTYC_NOCHANGE	No change; leave as is.										
PRTYC_IDLETIME	Idle-time.										
PRTYC_REGULAR	Regular.										
PRTYC_TIMECRITICAL	Time-critical.										
<i>sChange</i>	Specifies the change in priority to apply to the current priority level of the process. It can be any value from -31 to +31.										
<i>id</i>	Specifies a process or thread identifier, depending on the value of the <i>fScope</i> parameter. If <i>fScope</i> is PRTYS_PROCESS, the function sets 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. The error value may be one of the following:

- ERROR_INVALID_PCLASS
The priority class is invalid.
- ERROR_INVALID_PDELTA
The change in priority level is invalid.
- ERROR_INVALID_PROCID
The process identifier is invalid or non-existent.
- ERROR_INVALID_SCOPE
The scope must indicate either a thread or a process.
- ERROR_INVALID_THREADID
The thread identifier is invalid or non-existent.
- ERROR_NOT_DESCENDANT
Only a descendant's priority may be changed.

See Also

DosEnterCritSec, DosGetInfoSeg, DosGetPrty

- **USHORT DosSetSession(*idSession*, *pstsdata*)**
USHORT *idSession*; session identifier
PSTATUSDATA *pstsdata*; session status data

The **DosSetSession** function sets the status of a child session.

Parameter	Description
<i>idSession</i>	Specifies the identifier of the target session. The identifier must have been previously created by using the DosStartSession function.
<i>pstsdata</i>	Points to a STATUSDATA structure that contains the session status data. For a full description, see the following "Structures" section.

Return Value

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

Structures

The **STATUSDATA** structure pointed to by the *pstsdata* parameter has the following form:

```
typedef struct _STATUSDATA {
    USHORT cb;
    USHORT SelectInd;
    USHORT BindInd;
} STATUSDATA;
```

Field	Description								
cb	Specifies the length in bytes of the data structure.								
SelectInd	Specifies whether the target session should be set as selectable or non-selectable. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Leave current setting unchanged.</td></tr><tr><td>0x0001</td><td>Set as selectable.</td></tr><tr><td>0x0002</td><td>Set as non-selectable.</td></tr></table>	Value	Meaning	0x0000	Leave current setting unchanged.	0x0001	Set as selectable.	0x0002	Set as non-selectable.
Value	Meaning								
0x0000	Leave current setting unchanged.								
0x0001	Set as selectable.								
0x0002	Set as non-selectable.								
BindInd	Specifies which session to bring to the foreground the next time the parent session is selected. It has one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Leave current setting unchanged.</td></tr><tr><td>0x0001</td><td>A bond between the parent session and the child session is established. The child session is brought to the foreground the next time the parent session is selected. If the child session is selected, the child session is brought to the foreground.</td></tr><tr><td>0x0002</td><td>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 if the child is selected. Any bond previously established with the specified child session is broken.</td></tr></table>	Value	Meaning	0x0000	Leave current setting unchanged.	0x0001	A bond between the parent session and the child session is established. The child session is brought to the foreground the next time the parent session is selected. If the child session is selected, the child session is brought to the foreground.	0x0002	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 if the child is selected. Any bond previously established with the specified child session is broken.
Value	Meaning								
0x0000	Leave current setting unchanged.								
0x0001	A bond between the parent session and the child session is established. The child session is brought to the foreground the next time the parent session is selected. If the child session is selected, the child session is brought to the foreground.								
0x0002	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 if the child is selected. Any bond previously established with the specified child session is broken.								

Comments

The **DosSetSession** function sets/resets one or both of the following parameters related to a child session:

Selectable/non-selectable. This parameter allows a parent session to set one of its child sessions as either selectable or non-selectable.

Bind/no bind. This parameter allows a parent session to bind one of its child sessions to itself; that is, if the operator subsequently selects the parent session from the shell menu, the child session is brought to the foreground. If the parent session calls the **DosSelectSession** function and specifies its own session, the child session is brought to the foreground.

The parameters may be set individually. Either parameter can be changed without affecting the current setting of the other. The parameters affect selections made by the operator from the shell switch list. They do not affect selections made by the parent session. When a parent session selects its own session, its own session is brought to the foreground even if a bind is in effect. When a parent session selects a child session, the child session is brought to the foreground even if the parent session has previously set the child session to non-selectable.

The **DosSetSession** function can only be issued by a parent session for a child session. Neither the parent session itself nor any grandchild session can be the target of this call. Also, **DosSetSession** can only be used to change the status of child sessions that were originally started by the caller with the **DosStartSession** function specifying the **Related** field equal to one. The **DosSetSession** function cannot be used to change the status of sessions started as independent sessions.

A bond established between a parent session and a child session can be broken by reissuing the **DosSetSession** function and specifying either **BindInd** = 2 to break the bond, or **BindInd** = 1 to establish a bond with a different child session. In this case the bond with the previous child is broken.

Suppose a bond is established between session A and its immediate child session, B, and suppose another bond is established between session B and its immediate child session, C. If the user selects session A, session C is brought to the foreground. Assume that a bond is established between session A and its immediate child session B, and assume that session B is non-selectable. The user cannot select session B directly; if session A is selected, session B is brought to the foreground.

A parent session may be running in either the foreground or background when **DosSetSession** is called.

The **DosSetSession** function may be called only by the process which originally started the session specified by the *idSession* parameter through the **DosStartSession** function.

See Also

DosSelectSession, DosStartSession, DosStopSession

■ **USHORT DosSetSigHandler**(*pfnSigHandler*, *pfnPrev*, *pfAction*,
fAction, *usSigNumber*)
PFNSIGHANDLER *pfnSigHandler*; pointer to signal handler function
PFNSIGHANDLER FAR * *pfnPrev*; pointer to previous address
PUSHORT *pfAction*; pointer to variable for previous action
USHORT *fAction*; type of request
USHORT *usSigNumber*; signal number

The **DosSetSigHandler** function installs or removes a signal handler for the 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.

Parameter	Description
<i>pfnSigHandler</i>	Points to the signal handler. It must be the address of the function that receives control when the given signal occurs. For a full description, see the following “Comments” section.
<i>pfnPrev</i>	Points to a variable that receives the address of the previous signal handler.
<i>pfAction</i>	Points to a variable that receives the <i>fAction</i> parameter of the previous signal handler. This parameter can be a value from 0 to 3.
<i>fAction</i>	Specifies the type of request. It can be one of the following values:
Value	Meaning
0x0000	Install the system default action.
0x0001	Ignore the signal.
0x0002	Install the specified signal handler for the given signal.

usSigNumber 0x0003 Disallow process flag signals from other processes.
 0x0004 Reset the specified signal without affecting the disposition of the signal.
 Specifies the signal number. It can be one of the following values:

Value	Meaning
SIG_ CTRLC	CONTROL+C (signal interrupt, SIGINTR).
SIG_ BROKENPIPE	Process broken pipe
SIG_ KILLPROCESS	Program terminated (signal terminate, SIGTERM).
SIG_ CTRLBREAK	CONTROL+BREAK (signal break, SIGBREAK).
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. The error value may be one of the following:

ERROR_INVALID_FUNCTION
 The action specified is invalid.
 ERROR_INVALID_SIGNAL_NUMBER

Comments

The **DosSetSigHandler** function installs the signal handler that the system calls whenever the corresponding signal occurs. The signal handler is a function that can carry out tasks, such as cleaning up files, in response to a signal. 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 CONTROL+C or CONTROL+BREAK keys, when the process ends, or when another process calls the **DosProcessFlag** function. By default, the CONTROL+C, CONTROL+BREAK, and end-of-process signals terminate the process.

The **DosSetSigHandler** function copies the address of the previous signal handler to the variable pointed to by the *pfnPrev* parameter, and copies the corresponding action to the variable pointed to by the *pfAction* parameter. The new signal handler can use this address and action to pass the signal handler through a chain of previous handlers. It can also use the previous address and action to restore the previous handler.

The **DosSetSigHandler** function acknowledges a signal and re-enables it for subsequent input if the *fAction* parameter is set to 0x0004. 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 DosProcessFlag if appropriate */
USHORT usSigNum; /* Signal number being processed */
{
    .
    .
    .
    return;
}
```

Parameter	Description
<i>usSigArg</i>	Specifies the signal argument passed by the process that sends the process flag signal.
<i>usSigNum</i>	Specifies the signal number. It can be any of the values listed for the <i>usSigNumber</i> parameter.

When a signal occurs, the system calls the corresponding signal handler. The handler can then carry 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. The function may also use the **DosExit** function to terminate the process.

For assembly-language signal handlers, all registers other than **CS**, **IP**, **SS**, **SP**, and flags contain the same values that they contained at the time the signal was received. The handler may exit by executing an intersegment return instruction; execution resumes where it was interrupted, and all registers are restored to their values at the time of the interruption.

Family API Restrictions

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

- Only the signal-interrupt (SIGINTR) and signal-break (SIGBREAK) signals are available, therefore **DosSetSigHandler** may be used to install signal handlers for only these signals.
- The SIGINTR and SIGBREAK signals are treated as the same signal, so the function only accepts the SIG_CTRLC value when setting these signals.

See Also

DosCreateThread, **DosFlagProcess**, **DosHoldSignal**

■ **USHORT DosSetVec**(*usVecNum*, *pfFunction*, *ppfnPrev*)
USHORT *usVecNum*; type of exception
PFN *pfFunction*; pointer to function
PPFN *ppfnPrev*; pointer to variable for previous function's address

The **DosSetVec** function installs or removes an exception handler for the exception specified by the *usVecNum* parameter. An exception is a program error, such as dividing by zero or when an out-of-bounds exception occurs, that causes the system to pass control to the exception handler. The exception handler is an assembly-language routine that carries out tasks to correct errors or to clean up programs before terminating. The system calls the exception handler whenever the given exception occurs. If a process does not install its own exception handler, the default exception handler terminates the process.

The **DosSetVec** function copies the address of any previously registered exception handler to the variable pointed to by the *ppfnPrev* parameter. The new exception handler can use this address to chain exception handling through all previous handlers. It can also use this address to restore the previous handler.

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

Parameter	Description
<i>usVecNum</i>	Specifies the number of the exception vector. It can be one of the following values:

	Value	Meaning
	0x0000	Division by zero.
	0x0004	Overflow.
	0x0005	Out of bounds.
	0x0006	Invalid opcode.
	0x0007	Processor extension not available.
	0x0010	Processor extension error.
<i>pfnFunction</i>	Points to the exception handler. It must be the address of the routine that receives control on the specified exception, or it must be zero to direct DosSetVec to remove the current exception handler. For a full description, see the following “Comments” section.	
<i>ppfnPrev</i>	Points to a variable that receives the address of the previous exception handler.	

Return Value

The return value is zero if the function is successful. Otherwise, it is an error value. The error value 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 handler returns control, it must use the **reti** (return from interrupt) instruction.

If the **DosSetVec** function is used to install an exception handler for vector 0x0007 (processor extension not available), the function sets the machine status word (MSW) to indicate that no NPX 287 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 0x0007, the function sets the machine status word to reflect the true state of the hardware.

Family API Restrictions

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

- The *usVecNum* parameter may not be set to 0x0007 since this exception is not raised in machines using the 8088 or 8086 microprocessor.

See Also

DosDevConfig, DosError

■ **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 are written correctly to the disk whenever a process writes to a disk file.

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

Parameter	Description
<i>fVerify</i>	Specifies whether data verification is enabled. If the <i>fVerify</i> parameter is FALSE, verification is disabled. If it is TRUE, verification is enabled.

Return Value

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

ERROR_INVALID_VERIFY_SWITCH

Comments

Although errors when writing to a disk file are very rare, this function has been provided for applications that wish to verify the proper recording of critical data.

See Also

DosQVerify

- **USHORT DosSleep(*ulTime*)**
ULONG *ulTime*; amount of time to wait

The **DosSleep** function causes the current thread to wait for a specified interval of time or, if the requested interval is zero, to give up the remainder of the current time slice. The actual time the thread waits can be off by a clock tick or two, depending on the execution status of the other threads running in the system. If the specified time interval is zero, the process will forego the remainder of its CPU time slice but will be scheduled normally for its next time slice. Otherwise, the time is given in milliseconds, rounded up to the resolution of the scheduler clock.

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

Parameter	Description
<i>ulTime</i>	Specifies the number of milliseconds (rounded up to the next clock tick) that the thread waits.

Return Value

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

ERROR_TS_WAKEUP

The function returns before the specified time has elapsed.

Comments

For short time intervals, the rounding-up process combined with the thread-priority interactions can cause the waiting interval to be longer than the requested time. Also, when a process continues after suspension, it is scheduled for execution, although that execution could be delayed by hardware interrupts or by another thread running at a higher priority. In no case should the **DosSleep** function be substituted for a real-time clock because the rounding of the sleep interval will cause accumulative errors.

Example

This example sets up a loop that sleeps for one second and then retrieves the time and date:

```
for (;;) {
    DosSleep(1000L);          /* Waits for one second */
    DosGetDateTime(&dateTime); /* Retrieves the time and date */
    .
    .
    .
}
```


See Also

DosGetInfoSeg, DosTimerAsync, DosTimerStart,

- **USHORT DosStartSession**(*pstdata*, *pIdSession*, *ppid*)
PSTARTDATA *pstdata*; pointer to **STARTDATA** structure
PUSHORT *pIdSession*; pointer to variable for session identifier
PUSHORT *ppid*; pointer to variable for process identifier

The **DosStartSession** function starts another session and specifies the name of the program to be started in that session. This function creates either an independent session or a related session, depending on the value of the **Related** field in the **STARTDATA** structure.

Parameter	Description
<i>pstdata</i>	Points to a STARTDATA structure that contains data describing the session (screen group) to be started. For a full description, see the following “Structures” section.
<i>pIdSession</i>	Points to a variable that receives the identifier of the child session.
<i>ppid</i>	Points to a 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.

Structures

The **STARTDATA** structure pointed to by the *pstdata* parameter has the following form:

```
typedef struct _STARTDATA {
    USHORT cb;
    USHORT Related;
    USHORT FgBg;
    USHORT TraceOpt;
    PSZ    PgmTitle;
    PSZ    PgmName;
    PBYTE  PgmInputs;
    PBYTE  TermQ;
} STARTDATA;
```

Field	Description
cb	Specifies the length in bytes of the STARTDATA structure. The length must be set to 22 bytes.
Related	Specifies whether the session created is related to the calling session. If the Related field is 0x0000, the new session is an independent session (not related). If it is 0x0001, the new session is a child session (related).
FgBg	Specifies whether the new session should be started in the foreground or in the background. If the FgBg field is 0x0000, the session is started in the foreground. If it is 0x0001, the session is started in the background.
TraceOpt	Specifies whether the program started in the new session should be executed under conditions for tracing. If the TraceOpt field is 0x0000, there is no tracing. If it is 0x0001, 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 pathname of an MS OS/2 queue, or is equal to zero. This parameter is optional.

Comments

The MS OS/2 session manager writes a data element into the specified queue when the child session created as a result of the **DosStartSession** function terminates. A parent session calls the **DosReadQueue** function to be notified when a child session has terminated. The request word returned by **DosReadQueue** is zero, and the data-element format consists of two unsigned values: the session identifier and the result code.

The **DosReadQueue** function should be issued by the process that originally issued the **DosStartSession** request. This process is the only one that can address the notification data element. After reading and processing the data element, the caller must free the segment that contains the data element by using the **DosFreeSeg** function.

A related session or child session is created when the **Related** field is set to 0x0001. The child session is under direct control of the parent session. The parent session can specify a child session in a call to the **DosSelectSession**, **DosSetSession**, or **DosStopSession** function. For a related session, **DosStartSession** copies the session identifier of the child session to the variable pointed to by the *pldSession* parameter. It also copies the process identifier of the child process to the variable pointed to by the *ppid* parameter. This process identifier cannot be used with MS OS/2 functions, such as **DosSetPrty**, that require a parent process/child process relationship.

An independent session is not under the control of the starting session. It cannot be specified in a call to the **DosSelectSession**, **DosSetSession**, or **DosStopSession** function. The **DosStartSession** function does not copy session and process identifiers.

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**

■ **USHORT DosStopSession(*fScope*, *idSession*, *ulReserved*)**
USHORT *fScope*; scope of termination
USHORT *idSession*; session identifier
ULONG *ulReserved*; Must be zero

The **DosStopSession** function terminates a session previously started by the **DosStartSession** function.

Parameter	Description
<i>fScope</i>	Specifies whether the function stops all sessions or just the specified session. If the <i>fScope</i> parameter is 0x0000, the function stops only the specified session. If it is 0x0001, the function stops all sessions.
<i>idSession</i>	Specifies the identifier of the session to be stopped. The identifier must have been previously created by using the DosStartSession function. This parameter is ignored if the <i>fScope</i> parameter is set to 0x0001.
<i>ulReserved</i>	Specifies a reserved value. It 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 be issued only by a parent session for a child session. Neither the parent session itself nor any grandchild session can be the target of this function. However, if the child session specified in **DosStopSession** does have descendants, these sessions are terminated as well.

The only child sessions that **DosStopSession** can terminate are those that were originally started by the caller using the **DosStartSession** function with the **Related** field set to 0x0001. Sessions started as independent sessions cannot be stopped. The **DosStopSession** function can be issued only by the process that originally started the *idSession* parameter by using **DosStartSession**.

If a child session is in the foreground at the time it is stopped, the parent session becomes the foreground session. The **DosStopSession** function breaks any bond that exists between the parent session and the specified child session.

A parent session can be running in either the foreground or the background when **DosStopSession** is issued.

The process running in the session specified in the **DosStopSession** function can refuse to terminate. If this happens, **DosStopSession** returns zero. The only way to ensure that the target session has terminated is to wait for notification through the termination queue that is specified in the **DosStartSession** function.

See Also

DosSetSession, DosStartSession

■ **USHORT DosSubAlloc(sel, pusOffset, usSize)**
SEL sel; segment selector
PUSHORT pusOffset; pointer to variable for offset
USHORT usSize; size of allocation

The **DosSubAlloc** function allocates memory from a segment previously allocated by the **DosAllocSeg** or **DosAllocShrSeg** function and initialized by the **DosSubSet** function.

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

Parameter	Description
<i>sel</i>	Specifies the selector of the data segment from which the memory should be allocated.
<i>pusOffset</i>	Points to the variable that receives the offset to the allocated block.
<i>usSize</i>	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. The error value may be one of the following:

ERROR_DOSSUB_BADSIZE

The size parameter must be greater than zero.

ERROR_DOSSUB_NOMEM

No memory is available to satisfy request.

Comments

The *usSize* 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 *usSize* parameter does not need to be a multiple of 16.

Do not use **DosSubAlloc** without first allocating the memory segment and preparing it by using the **DosSubSet** function.

See Also

DosAllocSeg, **DosAllocShrSeg**, **DosSubFree**, **DosSubSet**

```

■ USHORT DosSubFree(sel, offBlock, cbBlock)
SEL sel;                segment selector
USHORT offBlock;        block offset
USHORT cbBlock;         size of block

```

The **DosSubFree** function frees memory previously allocated by the **DosSubAlloc** function.

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

Parameter	Description
<i>sel</i>	Specifies the selector of the data segment from which the memory should be freed.
<i>offBlock</i>	Specifies the offset of the memory block to be freed. The offset must have been previously created by using the DosSubAlloc function.
<i>cbBlock</i>	Specifies the size in bytes of the block to be freed.

Return Value

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

ERROR_DOSSUB_BADSIZE

The size parameter must be greater than zero.

ERROR_DOSSUB_OVERLAP

This block and another overlap.

See Also

DosAllocSeg, **DosSubAlloc**, **DosSubSet**

■ **USHORT DosSubSet**(*sel*, *fFlags*, *cbSeg*)

SEL <i>sel</i> ;	segment selector
USHORT <i>fFlags</i> ;	parameter flags
USHORT <i>cbSeg</i> ;	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.

Parameter	Description
<i>sel</i>	Specifies the selector of the target data segment.
<i>fFlags</i>	Specifies whether to initialize the segment or increase its size. If the <i>fFlags</i> parameter is 0x0001, the function initializes the segment. If it is 0x0000, the function changes the segment size.
<i>cbSeg</i>	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. The error value may be one of the following:

ERROR_DOSSUB_BADFLAG

The *fFlags* parameter is invalid.

ERROR_DOSSUB_BADSIZE

ERROR_DOSSUB_SHRINK

Cannot reduce the segment allocated by **DosSubAlloc**.

Comments

If the *fFlags* parameter is 0x0001, **DosSubSet** initializes the segment so that the **DosSubAlloc** function can be used to allocate memory blocks in the segment. The segment must have been previously allocated by using the **DosAllocSeg** or **DosAllocShrSeg** function.

If the *fFlags* parameter is 0x0000, **DosSubSet** changes the size of the segment to the number of bytes specified by the *cbSeg* parameter. If the size is greater than the previous size, the **DosReallocSeg** function must be called before **DosSubSet**. Failure to call **DosSubSet** after changing the size of a segment with **DosReallocSeg** can yield unpredictable results.

When changing a segment size, the *cbSeg* parameter must be a multiple of 4 bytes 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, the *cbSeg* parameter equal to zero indicates that the segment is 64K, but in the **DosSubAlloc** and **DosSubFree** functions, the *cbSeg* parameter equal to zero is an error.

See Also

DosAllocSeg, **DosAllocShrSeg**, **DosReallocSeg**, **DosSubAlloc**,
DosSubFree

■ **USHORT DosSuspendThread(*tid*)**

TID *tid*; thread identifier of thread to suspend

The **DosSuspendThread** function temporarily suspends the execution of a thread until a call to the **DosResumeThread** function is made for that thread identifier.

DosSuspendThread

Parameter	Description
<i>tid</i>	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. The error value may be the following:

ERROR_INVALID_THREADID

Comments

The specified thread may not be suspended immediately if it has called as system function which has some system resources locked that must be freed first. Once the thread is suspended, it will not be allowed to execute until a corresponding **DosResumeThread** function is issued.

A thread can suspend threads only within its process.

See Also

DosCreateThread, **DosEnterCritSec**, **DosResumeThread**

- **USHORT DosSystemService(*usCategory*, *pvoidRequest*, *pvoidResponse*)**
USHORT *usCategory*; service category
PVOID *pvoidRequest*; pointer to request packet
PVOID *pvoidResponse*; pointer to response packet

The **DosSystemService** function allows a process to request MS OS/2 to perform special functions, such as notifying the process of certain events that occur within the system.

Parameter	Description								
<i>usCategory</i>	Specifies the function requested or the event being waited for. The function type is one of the following values:								
<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Session manager services</td></tr><tr><td>0x0001</td><td>Hard-error services</td></tr><tr><td>0x0002</td><td>Print-screen services</td></tr></table>		Value	Meaning	0x0000	Session manager services	0x0001	Hard-error services	0x0002	Print-screen services
Value	Meaning								
0x0000	Session manager services								
0x0001	Hard-error services								
0x0002	Print-screen services								

pvoidRequest Points to a structure that contains information about the requested function. The request-packet structure has one of two formats, based on the function type. The formats of the request-packet structures are described in the following “Structures” section.

pvoidResponse Points to a structure that contains return information about the requested function. The response-packet structure has one of three formats, based on the function type. The formats of the response-packet structures are described in the following “Structures” section.

Return Value

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

ERROR_INVALID_FUNCTION

An invalid or in-use function was requested.

Structures

Each *usCategory* parameter has its own data structures for the request and response packets. The request and response packets for each *usCategory* parameter are described together.

If the *usCategory* parameter is 0x0000, the **SMService** field of the request-packet structure specifies the function to perform. The request packet may contain additional parameters, based on the specified function. The request-packet structure has a form identical to the following structure:

```
struct {
    USHORT SMSService;
    USHORT SMSSubFunction;
    USHORT Item;
};
```

Field	Description
SMSService	Specifies the type of service requested. If this field is 0x0000, the function waits until the session-manager hot key is pressed. If it is 0x0001, the function performs a freeze or thaw subfunction.
SMSSubFunction	Specifies the freeze or thaw subfunction performed when SMSService is 0x0001. It is one of the following values:

	Value	Meaning
	0x0000	Freeze process.
	0x0001	Thaw process.
	0x0002	Freeze screen group.
	0x0003	Thaw screen group.
	0x0004	Get real-mode process identifier.
Item	Specifies the item to be acted on. It is one of the following values:	
	Value	Meaning
	0-1	Process identifier
	2-3	Screen-group number

When the **SMSERVICE** field is 0x0000 (the function waits until the session-manager hot key is pressed), the response-packet structure has a form identical to the following structure:

```
struct {
    USHORT HotKeyDevice;
    USHORT TimeStamp;
};
```

Field	Description
HotKeyDevice	Specifies the type of hot-key device in use. If the HotKeyDevice field is 0x0000, the mouse is in use. If it is 0x0001, the keyboard is in use.
TimeStamp	If a mouse is in use as the hot key, the TimeStamp field specifies the time of a mouse-device hot-key event in two bytes. The first byte contains the time in seconds, the second contains the time in hundredths of seconds. If a keyboard is in use as the hot key, this field specifies a hot-key identifier.

If the session-manager hot key is pressed again, before a prior hot key has been serviced and before **DosSystemService** is called again, notification remains pending in the kernel.

If the *usCategory* parameter is 0x0001, the request-packet structure contains the action to take in response to a hard error returned on a previous **DosSystemService** service category 1 request. This request-packet structure accompanies the **DosSystemService** function to wait on the next

hard error. If there was no hard error on the first system function that specified service category 1, the request-packet structure is not referenced.

The request-packet structure has a form identical to the following structure:

```
struct {
    USHORT HEService;
};
```

Field	Description
HEService	Specifies the action to take in response to prior hard-error notification. It can be one of the following values:
Value	Meaning
0x0000	Return immediately from function that caused error.
0x0001	Terminate process that generated error.
0x0002	Retry operation that caused error.
0x0003	Ignore error.

The response-packet structure has a form identical to the following structure:

```
struct {
    USHORT SGNumber
    USHORT ProcessID
    USHORT MsgNumber
    USHORT ValidResponse
    USHORT DefaultResponse
    USHORT NumStrings
    USHORT BUFLen
    CHAR Buffer[BUFLen];
};
```

Field	Description
SGNumber	Specifies the number of the screen group where the error occurred.
ProcessID	Specifies the process identifier of the process that caused the error.
MsgNumber	Specifies the error-message number to be passed to the DosGetMessage function.
ValidResponse	Specifies valid responses. It is one of the following values:

Value	Meaning
0x0001	Return immediately from function that caused error.
0x0002	Terminate process that generated error.
0x0004	Retry operation that caused error.
0x0008	Ignore error.

DefaultResponse

Specifies the default response. It is one of the following values:

Value	Meaning
0x0001	Return immediately from function that caused error.
0x0002	Terminate process that generated error.
0x0004	Retry operation that caused error.
0x0008	Ignore error.

NumStrings

Specifies the number of null-terminated text strings returned in the **Buffer** field.

BUFLEN

Specifies the size in bytes of the text buffer specified by the **Buffer** field. This value is passed as input when the **DosSystemService** function is called and is returned unmodified.

Buffer[BUFLEN]

Specifies a character array that receives the null-terminated text strings (the array contains the variable insertion text to be passed to **DosGetMessage** for this error-message number). If there is more than one text string, the strings must immediately follow one another with no intervening bytes. The order in which the text strings are placed in the buffer must be the same as the order in which the text strings are required in the error value. The maximum text-buffer size required is 256 bytes.

If any additional hard errors are detected before the system process servicing a previous hard error has called the **DosSystemService** function again, notification remains pending in the kernel.

If the *usCategory* parameter is 0x0002 (wait for the first/next SHIFT+PRINTSCREEN or CONTROL+PRINTSCREEN key combination), there is no request-packet structure.

The request-packet structure has a form identical to the following structure:

```
struct {
    USHORT KeyStroke;
};
```

Field	Description
KeyStroke	Specifies the key combination that was pressed. If the KeyStroke field is 0x0000, SHIFT+PRINTSCREEN was pressed. If it is 0x0001, CONTROL+PRINTSCREEN was pressed.

Comments

The **DosSystemService** function allows one process, such as the session manager, to perform special functions or to block execution until a particular event occurs. When the kernel completes the requested function, or the event being waited for has occurred, the kernel returns from the **DosSystemService** function.

For each function code, only one process is allowed to call **DosSystemService**. This restriction is enforced by the kernel, which records the process identifier when the initial **DosSystemService** function for a particular function is called.

See Also

DosGetMessage

- **USHORT DosTimerAsync(*ulTime*, *hsem*, *phtimer*)**
ULONG *ulTime*; interval size
HSEM *hsem*; system-semaphore handle
PHTIMER *phtimer*; pointer to variable for timer handle

The **DosTimerAsync** function creates a timer and copies a timer handle to the variable pointed to by the *phtimer* parameter. The timer counts the number of milliseconds specified by the *ulTime* parameter, then clears the semaphore identified by the *hsem* parameter.

The timer runs asynchronously; that is, while the timer continues to count the time, the 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 set the semaphore by using the **DosSemSet** function before calling **DosTimerAsync**.

The timer handle specified by the *phtimer* parameter identifies the timer and can be used in the **DosTimerStop** function to cancel the timer.

Parameter	Description
<i>ulTime</i>	Specifies the time in milliseconds before the semaphore is cleared.
<i>hsem</i>	Identifies the system semaphore that signals the end of the timer. The handle must have been previously created by using the DosCreateSem function.
<i>phtimer</i>	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. The error value may be one of the following:

ERROR_TS_NOTIMER

ERROR_TS_SEMHANDLE

The semaphore handle is invalid.

Comments

The **DosTimerAsync** function is similar to the **DosSleep** function except that **DosTimerAsync** returns immediately; **DosSleep** returns only after the given time has elapsed.

See Also

DosSemSet, **DosSleep**, **DosTimerStart**, **DosTimerStop**

- **USHORT** **DosTimerStart**(*ulTime*, *hsem*, *phtimer*)
ULONG *ulTime*; interval size
HSEM *hsem*; system-semaphore handle
PHTIMER *phtimer*; pointer to variable for timer handle

The **DosTimerStart** function creates a timer and copies a timer handle to the variable pointed to by the *phtimer* parameter. The timer counts the

number of milliseconds specified by the *ulTime* parameter, then clears the semaphore identified by the *hsem* parameter. It repeats this process continually, counting the time and clearing the semaphore, until the process stops it by using the **DosTimerStop** function. The timer handle identifies the timer and must be used in the **DosTimerStop** function to cancel the timer.

The timer runs asynchronously; that is, while the timer continues to count 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 set the semaphore by using the **DosSemSet** function before calling **DosTimerAsync**.

Parameter	Description
<i>ulTime</i>	Specifies the time in milliseconds before the semaphore is cleared.
<i>hsem</i>	Identifies the system semaphore that signals the end of the timer. The handle must have been previously created by using the DosCreateSem function.
<i>phtimer</i>	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. The error value may be one of the following:

ERROR_TS_NOTIMER

No timers are available.

ERROR_TS_SEMHANDLE

The semaphore handle is invalid.

Comments

If necessary, the **DosTimerStart** function rounds up the *ulTime* parameter to the next clock tick.

If some cases, the timer may clear the semaphore several times before a given process waiting for it resumes execution. If the process requires an accurate count of elapsed time, it should retrieve the current system time from the global information segment before and after waiting for the semaphore, and then compare the times.

See Also

DosGetInfoSeg, DosSemSet, DosTimerStop

- **USHORT DosTimerStop(*htimer*)**
HTIMER *htimer*; timer handle

The **DosTimerStop** function stops the timer identified by the *htimer* parameter. The timer stops only if it is still counting the time. Otherwise, the function has no effect.

Parameter	Description
<i>htimer</i>	Identifies the timer to be stopped. The handle must have been previously created 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. The error value may be the following:

ERROR_TS_HANDLE
The timer handle is invalid.

Comments

When the **DosTimerStop** function stops a timer, it does not clear the semaphore that corresponds to the given timer. 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

- **USHORT DosUnlockSeg(*sel*)**
SEL *sel*; selector of segment to be unlocked

The **DosUnlockSeg** function unlocks a discardable segment. Once a segment is unlocked, the system may discard it to make space available for other segments.

Parameter	Description
<i>sel</i>	Specifies the selector of the segment to be unlocked.

Return Value

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

Comments

The **DosUnlockSeg** function applies only to segments that have been allocated by using the **DosAllocSeg** function with the *AllocFlags* parameter set to **SEG_DISCARDABLE**.

The **DosLockSeg** and **DosUnlockSeg** functions may be nested. For example, 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**

■ **USHORT** **DosWrite**(*hf*, *pvoidBuf*, *cbBuf*, *pcbBytesWritten*)
HFILE *hf*; file handle
PVOID *pvoidBuf*; pointer to buffer
USHORT *cbBuf*; length of the buffer
PUSHORT *pcbBytesWritten*; pointer to variable for bytes written

The **DosWrite** function writes one or more bytes of data to the file identified by the *hf* parameter. The *cbBuf* parameter specifies the number of bytes to write to the file. The function may write less bytes if there is not enough space on the disk. The **DosWrite** function copies the actual number of bytes written to the variable pointed to by the *pcbBytesWritten* parameter. The *cbBuf* parameter can be zero without causing an error; that is, writing no bytes is acceptable.

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

Parameter	Description
<i>hf</i>	Identifies the file that receives the data. The handle must have been previously created by using the DosOpen function.
<i>pvoidBuf</i>	Points to the buffer that contains the data to be written.
<i>cbBuf</i>	Specifies the number of bytes to be written.
<i>pcbBytesWritten</i>	Points to the variable that receives the number of bytes written.

Return Value

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

ERROR_ACCESS_DENIED

Cannot write to a file that was opened as read-only.

ERROR_BROKEN_PIPE

ERROR_INVALID_HANDLE

ERROR_LOCK_VIOLATION

ERROR_NOT_DOS_DISK

Comments

The **DosWrite** function writes bytes starting at the current file-pointer position. If desired, the file-pointer position can be changed by using the **DosChgFilePtr** function.

If the specified file has been opened using the write-through flag, **DosWrite** also 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 operates most efficiently when the number of bytes it writes is a multiple of the bytes-per-sector size of the disk or a multiple of whatever size the internal file buffer is (this depends on the device). The function writes directly to the disk without copying the data to an internal file buffer. The **DosQFSInfo** function can be used to retrieve the bytes-per-sector value for a disk.

The **DosWrite** function returns an error if an attempt is made to write to a file that was opened with read-only access.

Example

This example opens the file *abc* and calls the **DosWrite** function to write the contents of the *cbBuf* variable to the file:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesWritten;
if (!DosOpen("abc", &hf, &usAction, OL, O, Ox10, Ox41, OL)) {
    DosWrite(hf,                                /* file handle */
             abBuf,                             /* buffer address */
             sizeof(abBuf),                    /* buffer size */
             &cbBytesWritten);                 /* address of bytes written */
}
```

See Also

DosChgFilePtr, DosOpen, DosRead, DosWriteAsync

■ **USHORT DosWriteAsync**(*hf, hsemRam, pusErrCode, pvoidBuf, cbBuf, pcbBytesWritten*)

HFILE <i>hf</i> ;	file handle
PULONG <i>hsemRam</i> ;	pointer to RAM semaphore
PUSHORT <i>pusErrCode</i> ;	pointer to variable for error value
PVOID <i>pvoidBuf</i> ;	pointer to buffer
USHORT <i>cbBuf</i> ;	length of buffer
PUSHORT <i>pcbBytesWritten</i> ;	pointer to variable for bytes written

The **DosWriteAsync** function writes one or more bytes of data to the file identified by the *hf* parameter. 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.

The *cbBuf* parameter specifies the number of bytes to write to the file. The function may write less bytes if there is not enough space on the disk. The **DosWriteAsync** function copies the actual number of bytes written to the variable pointed to by the *pcbBytesWritten* parameter. The *cbBuf* parameter can be zero without causing an error; that is, writing no bytes is acceptable.

When **DosWriteAsync** 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 set the semaphore by using the **DosSemSet** function before calling **DosWriteAsync**.

If **DosWriteAsync** encounters an error while writing data, it copies the error value to the variable pointed to by the *pusErrCode* parameter. The possible error values are identical to those returned by the **DosWrite** function.

Parameter	Description
<i>hf</i>	Identifies the file that receives the data. The handle must have been previously created by using the DosOpen function.
<i>hsemRam</i>	Points to the RAM semaphore that indicates when the function has finished reading the data.
<i>pusErrCode</i>	Points to the variable that receives any error value the function may generate.
<i>pvoidBuf</i>	Points to the buffer that contains the data to be written.
<i>cbBuf</i>	Specifies the number of bytes to be written.
<i>pcbBytesWritten</i>	Points to the variable that receives the number of bytes written.

Return Value

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

ERROR_ACCESS_DENIED

Cannot write to a file that was opened as read-only.

ERROR_BROKEN_PIPE

ERROR_INVALID_HANDLE

ERROR_LOCK_VIOLATION

ERROR_NO_PROC_SLOTS

Cannot create a thread for this operation.

ERROR_NOT_DOS_DISK

Comments

The **DosWriteAsync** function writes bytes starting at the current file-pointer position. If desired, the file-pointer position can be changed by using the **DosChgFilePtr** function.

If the specified file has been opened using the write-through flag, **DosWriteAsync** also 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 **DosWriteAsync** function operates most efficiently when the number of bytes it writes is a multiple of the bytes-per-sector size of the disk or a multiple of whatever size the internal file buffer is (this depends on the

device). The function writes directly to the disk without copying the data to an internal file buffer. The **DosQFSInfo** function can be used to retrieve the bytes-per-sector value for a disk.

The **DosWriteAsync** function returns an error if an attempt is made to write to a file that was opened with read-only access.

Example

This example opens the file *abc.txt*, sets a RAM semaphore, and calls the **DosWriteAsync** function to write out the contents of the buffer *chBuf*. When any additional processing is completed, the example calls the **DosSemWait** function to wait until **DosWriteAsync** has completed writing to the file:

```
ULONG hsemWrite = 0;
BYTE abBuf[1024];
HFILE hf;
USHORT usAction, cbBytesWritten;
USHORT usWriteReturn;
DosOpen("abc", &hf, &usAction, OL, O, Ox10, Ox41, OL);
DosSemSet(&hsemWrite);           /* Sets the semaphore */
DosWriteAsync(hf,                 /* file handle */
               &hsemWrite,        /* semaphore address */
               &usWriteReturn,     /* 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 */
```

See Also

DosChgFilePtr, **DosOpen**, **DosReadAsync**, **DosSemSet**, **DosWrite**

■ **USHORT DosWriteQueue**(*hqueue*, *usRequest*, *cbBuf*,
pbBuf, *usPriority*)

HQUEUE <i>hqueue</i> ;	handle of queue to send to
USHORT <i>usRequest</i> ;	request-identification data
USHORT <i>cbBuf</i> ;	length of element being added
PBYTE <i>pbBuf</i> ;	pointer to element being added
USHORT <i>usPriority</i> ;	priority of element being added

The **DosWriteQueue** function writes an element to the given queue. The function copies the element from the buffer pointed to by the *pbBuf* parameter to the queue and assigns it the priority given by the *usPriority* parameter. The element is added to the beginning, end, or other position in the

queue, depending the priority method of the given queue. After the element is written, the process that owns the queue may read the element by using the **DosPeekQueue** or **DosReadQueue** function.

Parameter	Description
<i>hqueue</i>	Identifies the queue to be written to. The handle must have been previously created or opened by using the DosCreateQueue or DosOpenQueue function.
<i>usRequest</i>	Specifies a program-supplied event code. MS OS/2 does not use this field and reserves it for any use a program may want to make of it. The queue owner can retrieve this value by using the DosPeekQueue or DosReadQueue function.
<i>cbBuf</i>	Specifies the number of bytes to be copied to the buffer pointed to by the <i>pbBuf</i> parameter.
<i>pbBuf</i>	Points to the buffer that contains the element to be written to the queue.
<i>usPriority</i>	Specifies the element priority. It can be any value from 0 to 15; 15 is the highest priority.

Return Value

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

ERROR_QUE_INVALID_HANDLE
The queue handle is invalid.

ERROR_QUE_NO_MEMORY
The queue segment is full.

Comments

The **DosWriteQueue** function returns an error value if the process that owns the queue has closed the queue.

If the queue owner has defined a semaphore for use in its notification when elements are added to the queue, the semaphore must be shared if it is a RAM semaphore. If it is a system semaphore, the writer must also have opened it.

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**

- **USHORT KbdCharIn**(*pkbciKeyInfo*, *fNoWait*, *hkbd*)
PKBDKEYINFO *pkbciKeyInfo*; pointer to buffer for keystroke
USHORT *fNoWait*; 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 keystroke information to the structure pointed to by the *pkbciKeyInfo* 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.

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

Parameter	Description
<i>pkbciKeyInfo</i>	Points to the KBDKEYINFO structure that receives the character data. For a full description, see the following “Structures” section.
<i>fNoWait</i>	Specifies whether to wait for data if none is available. If the <i>fNoWait</i> parameter is IO_WAIT , the function waits for a character if one is not available. If it is IO_NOWAIT , the function returns immediately if a character is not available. The fbStatus field in the KBDKEYINFO structure specifies whether a keystroke is actually received.
<i>hkbd</i>	Identifies the logical keyboard. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

ERROR_KBD_INVALID_IOWAIT
The *fNoWait* parameter is invalid.

Structures

The **KBDKEYINFO** structure pointed to by the *pkbciKeyInfo* parameter has the following form:


```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

Field	Description																				
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: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>A shift key is received (valid only in raw mode when shift report is on).</td></tr><tr><td>0x0020</td><td>Conversion requested.</td></tr><tr><td>0x0040</td><td>Final character received.</td></tr><tr><td>0x0080</td><td>Interim character received.</td></tr></table>	Value	Meaning	0x0001	A shift key is received (valid only in raw mode when shift report is on).	0x0020	Conversion requested.	0x0040	Final character received.	0x0080	Interim character received.										
Value	Meaning																				
0x0001	A shift key is received (valid only in raw mode when shift report is on).																				
0x0020	Conversion requested.																				
0x0040	Final character received.																				
0x0080	Interim character received.																				
bNlsShift	Specifies a reserved value. It must be zero.																				
fsState	Specifies the state of the shift keys. It can be any combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>SHIFT key up.</td></tr><tr><td>0x0001</td><td>Right SHIFT key down.</td></tr><tr><td>0x0002</td><td>Left SHIFT key down.</td></tr><tr><td>0x0004</td><td>Either CONTROL key down.</td></tr><tr><td>0x0008</td><td>Either ALT key down.</td></tr><tr><td>0x0010</td><td>SCROLL LOCK key turned on.</td></tr><tr><td>0x0020</td><td>NUMLOCK key turned on.</td></tr><tr><td>0x0040</td><td>CAPSLOCK key turned on.</td></tr><tr><td>0x0080</td><td>INSERT key turned on.</td></tr></table>	Value	Meaning	0x0000	SHIFT key up.	0x0001	Right SHIFT key down.	0x0002	Left SHIFT key down.	0x0004	Either CONTROL key down.	0x0008	Either ALT key down.	0x0010	SCROLL LOCK key turned on.	0x0020	NUMLOCK key turned on.	0x0040	CAPSLOCK key turned on.	0x0080	INSERT key turned on.
Value	Meaning																				
0x0000	SHIFT key up.																				
0x0001	Right SHIFT key down.																				
0x0002	Left SHIFT key down.																				
0x0004	Either CONTROL key down.																				
0x0008	Either ALT key down.																				
0x0010	SCROLL LOCK key turned on.																				
0x0020	NUMLOCK key turned on.																				
0x0040	CAPSLOCK key turned on.																				
0x0080	INSERT key turned on.																				

0x0100	Left CONTROL key down.
0x0200	Left ALT key down.
0x0400	Right CONTROL key down.
0x0800	Right ALT key down.
0x1000	SCROLL LOCK key down.
0x2000	NUMLOCK key down.
0x4000	CAPSLOCK key down.
0x8000	SYSREQ key down.

time Specifies the time stamp of the keystroke in milliseconds.

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 cooked mode, **KbdCharIn** retrieves keystroke information for each key pressed except shift keys and MS OS/2 control keys. If the keyboard is in raw mode, **KbdCharIn** retrieves keystroke information for any key pressed except shift keys. In most cases, a shift key is pressed together with other keys to create a single keystroke. In raw mode with shift report on, a shift key by itself creates a keystroke that can be retrieved by the **KbdCharIn** function. For more information on raw 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 **KeyData** 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. For more information on extended ASCII codes, see Appendix C, "Code Pages."

This function must be called twice to retrieve a double-byte character-set (DBCS) code. The **fbStatus** field of the **KeyData** structure is set to 0x0080 only if the interim character flag is set. For more information about getting and setting the interim character flags, see the **KbdGetStatus** and **KbdSetStatus** functions.

Family API 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 keyboard (handle 0).
- The **fbStatus** field may be 0x0000 or 0x0001.
- 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 kbcKeyInfo;
KbdCharIn(&kbcKeyInfo,                                /* buffer for data */
0,                                                    /* wait/no wait    */
0);                                                  /* keyboard handle */
VioWrtTTY(&kbcKeyInfo.chChar, 1, 0);
```

See Also

KbdPeek, **KbdStringIn**

■ **USHORT KbdClose(*hkbd*)**
HKBD *hkbd*; keyboard handle

The **KbdClose** function closes the specified logical keyboard. The function flushes any remaining keystrokes from the input buffer and automatically frees the focus (if the logical keyboard has it).

Parameter	Description
<i>hkbd</i>	Identifies the logical keyboard to be closed. The handle must have been previously opened by using the KbdOpen function.

KbdClose

Return Value

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

ERROR_KBD_INVALID_HANDLE

The specified keyboard handle is invalid.

Comments

The default keyboard cannot be closed. If the default keyboard (handle 0) is specified, the **KbdClose** function ignores the request.

Example

This example opens a logical keyboard and calls the **KbdClose** function to close it:

```
HKBD hkbd;  
KbdOpen(&hkbd);  
.  
.  
KbdClose(hkbd);
```

See Also

KbdFlushBuffer, KbdFreeFocus, KbdOpen

■ **USHORT KbdDeRegister(VOID)**

The **KbdDeRegister** function restores the default keyboard subsystem functions and releases any previously registered keyboard subsystem. The function restores the default keyboard subsystem for all processes in the current screen group.

This function has no parameters.

Return Value

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

ERROR_KBD_DEREGISTER

The **KbdDeRegister** function is not allowed.

Comments

Once a process registers a keyboard subsystem, no other process in the screen group may register a keyboard subsystem until the default subsystem is restored. Only the process registering a keyboard subsystem may call the **KbdDeRegister** function to restore the default subsystem.

See Also

KbdRegister

- **USHORT KbdFlushBuffer(*hkbd*)**
HKBD *hkbd*; keyboard handle

The **KbdFlushBuffer** function flushes the input buffer of the specified logical keyboard by removing all keystroke information. The **KbdFlushBuffer** function flushes the input buffer only if the specified logical keyboard has the focus or is the default keyboard.

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

Parameter	Description
<i>hkbd</i>	Identifies the logical keyboard to be flushed. It must have been previously opened by using the KbdOpen function.

Return Value

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

Family API 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 the **KbdFlushBuffer** function to clear any keystrokes in the input buffer:

```
HKBD hkbd;  
KbdOpen (&hkbd) ;  
.  
.  
KbdFlushBuffer (hkbd) ;
```

See Also

KbdCharIn

- **USHORT KbdFreeFocus(*hkbd*)**
HKBD *hkbd*; keyboard handle

The **KbdFreeFocus** function frees the focus from the specified logical keyboard. The focus is then available for use by other logical keyboards.

Parameter	Description
<i>hkbd</i>	Identifies the logical keyboard that loses the focus. The handle must have been previously opened 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, the system assigns the focus to the logical keyboard as soon as it is free. If more than one process is waiting, the system arbitrarily chooses a logical keyboard and assigns the focus. The other processes continue to wait until the focus is free.

Example

The following example frees a logical keyboard and, if other logical keyboards are waiting, the system assigns the focus to one of them. If no other logical keyboards are waiting, the physical keyboard uses the default keyboard:

```
HKBD hkbd;  
.  
.  
KbdFreeFocus (hkbd) ;
```

See Also

KbdGetFocus

■ **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 the system uses to translate keystrokes into character values. The **KbdGetCp** function copies the identifier to the variable pointed to by the *pIDCodePage* parameter.

Parameter	Description
<i>ulReserved</i>	Specifies a reserved value. It must be zero.
<i>pIDCodePage</i>	Points to the variable that receives the code-page identifier.
<i>hkbd</i>	Identifies the logical keyboard. The handle must have been previously opened 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 the system 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, "Code Pages."

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(OL,  
          &idCodePage,  
          0);  
/* Must be zero  
/* pointer to code-page identifier */  
/* keyboard handle */
```

See Also

DosGetCp, KbdSetCp

- **USHORT KbdGetFocus**(*fNoWait*, *hkbd*)
USHORT *fNoWait*; 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 in use 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. A process can wait by setting the *fNoWait* parameter to **IO_WAIT**. If more than one process waits for the focus, the order in which each receives the focus is arbitrary.

Parameter	Description
<i>fNoWait</i>	Specifies whether to wait for the focus. If this parameter is IO_WAIT , the function waits for the focus. If it is IO_NOWAIT , the function returns immediately without getting the focus.
<i>hkbd</i>	Identifies the logical keyboard that receives the focus. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

Example

This example opens a logical keyboard and calls the **KbdGetFocus** function to get the focus for the opened keyboard. Before an application can call functions such as **KbdCharIn** by using the opened keyboard handle, it must first set the focus to the new handle. Once the **KbdFreeFocus** function is called, the focus goes to any process that is waiting for it by using a call to the **KbdGetFocus** function. If no process is waiting, the focus returns to handle 0, the default keyboard:


```
HKBD hkbd;
KbdOpen (&hkbd) ;
KbdGetFocus (IO_WAIT, hkbd) ;
.
.
.
KbdFreeFocus (hkbd) ;
```

See Also

KbdFreeFocus

■ **USHORT KbdGetStatus**(*pkbstKbdInfo*, *hkbd*)
PKBDINFO *pkbstKbdInfo*; pointer to buffer for 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, turn-around character, interim character flags, and shift state.

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

Parameter	Description
<i>pkbstKbdInfo</i>	Points to the KBDINFO structure that receives the keyboard status. For a full description, see the following “Structures” section.
<i>hkbd</i>	Identifies the logical keyboard. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

ERROR_KBD_INVALID_LENGTH
The structure length is invalid.

Structures

The **KBDINFO** structure pointed to by the *pkbstKbdInfo* parameter has the following format:

```
typedef struct _KBDINFO {
    USHORT cb;
    USHORT fsMask;
    USHORT chTurnAround;
    USHORT fsInterim;
    USHORT fsState;
} KBDINFO;
```

Field	Description																
cb	Specifies the length in bytes of the KBDINFO structure. This field must be set to 0x000A before calling the function.																
fsMask	Specifies the current keyboard modes. It can be any combination of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>No states set.</td></tr> <tr> <td>0x0001</td><td>Echo mode turned on.</td></tr> <tr> <td>0x0002</td><td>Echo mode turned off.</td></tr> <tr> <td>0x0004</td><td>Raw mode turned on.</td></tr> <tr> <td>0x0008</td><td>Cooked mode turned on.</td></tr> <tr> <td>0x0080</td><td>Two-byte turn-around character. If not given, the turn-around character is one byte.</td></tr> <tr> <td>0x0100</td><td>Shift report turned on.</td></tr> </table> <p>Echo mode is either turned on or off, not both. Only one input mode, raw or cooked, can be turned on at any given time.</p>	Value	Meaning	0x0000	No states set.	0x0001	Echo mode turned on.	0x0002	Echo mode turned off.	0x0004	Raw mode turned on.	0x0008	Cooked mode turned on.	0x0080	Two-byte turn-around character. If not given, the turn-around character is one byte.	0x0100	Shift report turned on.
Value	Meaning																
0x0000	No states set.																
0x0001	Echo mode turned on.																
0x0002	Echo mode turned off.																
0x0004	Raw mode turned on.																
0x0008	Cooked mode turned on.																
0x0080	Two-byte turn-around character. If not given, the turn-around character is one byte.																
0x0100	Shift report turned on.																
chTurnAround	Specifies the turn-around character. If this field value includes 0x0080, the character is two-bytes packed in the low- and high-order bytes of this field. Otherwise, the character is a single byte in the low-order 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: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>SHIFT key up.</td></tr> <tr> <td>0x0001</td><td>Right SHIFT key down.</td></tr> </table>	Value	Meaning	0x0000	SHIFT key up.	0x0001	Right SHIFT key down.										
Value	Meaning																
0x0000	SHIFT key up.																
0x0001	Right SHIFT key down.																

0x0002	Left SHIFT key down.
0x0004	CONTROL key down.
0x0008	ALT key down.
0x0010	SCROLL LOCK key turned on.
0x0020	NUMLOCK key turned on.
0x0040	CAPSLOCK key turned on.
0x0080	INSERT key turned on.

Comments

Although the initial status of a logical keyboard depends on the system, the logical keyboard typically has echo and cooked modes turned on, and has a single-byte turn-around character whose character value corresponds to the ENTER key.

Family API 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);          /* status-buffer length */
KbdGetStatus(&kbstInfo, 0);
if (kbstInfo.fsMask & 0x0001) {          /* Checks echo bit      */
    VioWrtTTY("Echo is on\n\r", 12, 0);
```

See Also

KbdSetStatus

- **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 keyboard (or other operating-system) functions. The **KbdOpen** function initializes the logical keyboard to use the system default code page.

Parameter	Description
<i>phkbd</i>	Points to a variable that receives a logical keyboard handle.

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 from a handle (for example, the **DosRead** function) can use the handle retrieved by the **KbdOpen** function.

Example

This example calls the **KbdOpen** function to create and open a handle for a logical keyboard, and to initialize it to use the default system code page. Before you can access this logical keyboard you must call **KbdGetFocus** to get the focus:

```
HKBD hkbd;  
KbdOpen(&hkbd);  
KbdGetFocus(IO_WAIT, hkbd);
```

See Also

DosRead, **KbdClose**, **KbdGetFocus**

- **USHORT KbdPeek**(*pkbciKeyInfo*, *hkbd*)
PKBDKEYINFO *pkbciKeyInfo*; pointer to buffer for keystroke
HKBD *hkbd*; keyboard handle

The **KbdPeek** function retrieves character and scan-code information from a logical keyboard. The function copies keystroke information to the structure pointed to by the *pkbciKeyInfo* parameter. The keystroke information includes the character value of the key, scan code, keystroke status, state of the shift keys, and system time (in milliseconds) when the keystroke occurred.

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 cooked mode, **KbdPeek** retrieves keystroke information for each key pressed, except shift keys and MS OS/2 control keys. If the keyboard is in raw mode,

KbdPeek retrieves keystroke information for any key pressed, except shift keys. In most cases, a shift key is pressed together with other keys to create a single keystroke. In raw mode with shift report turned on, a shift key by itself creates a keystroke that can be retrieved by this function. For more information on raw mode and shift-report mode, see the **KbdSetStatus** function.

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

Parameter	Description
<i>pkbciKeyInfo</i>	Points to the KBDKEYINFO structure that receives the character data. For a full description, see the following “Structures” section.
<i>hkbd</i>	Identifies the logical keyboard. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

Structures

The **KBDKEYINFO** structure pointed to by the *pkbciKeyInfo* parameter has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR chChar;
    UCHAR chScan;
    UCHAR fbStatus;
    UCHAR bNlsShift;
    USHORT fsState;
    ULONG time;
} KBDKEYINFO;
```

Field	Description
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 code may have been modified during the translation process.
fbStatus	Specifies the state of the retrieved scan code. It can be any combination of the following values:

	<u>Value</u>	<u>Meaning</u>
	0x0001	A shift key is received (valid only in raw mode when shift report is turned on).
	0x0020	Conversion requested.
	0x0040	Final character received.
	0x0080	Interim character received.
bNlsShift		Specifies a reserved value. It is set to zero.
fsState		Specifies the state of the shift keys. It can be any combination of the following values:
	<u>Value</u>	<u>Meaning</u>
	0x0000	SHIFT key up.
	0x0001	Right SHIFT key down.
	0x0002	Left SHIFT key down.
	0x0004	Either CONTROL key down.
	0x0008	Either ALT key down.
	0x0010	SCROLL LOCK key turned on.
	0x0020	NUMLOCK key turned on.
	0x0040	CAPSLOCK key turned on.
	0x0080	INSERT key turned on.
	0x0100	Left CONTROL key down.
	0x0200	Left ALT key down.
	0x0400	Right CONTROL key down.
	0x0800	Right ALT key down.
	0x1000	SCROLL LOCK key down.
	0x2000	NUMLOCK key down.
	0x4000	CAPSLOCK key down.
	0x8000	SYSREQ key down.
time		Specifies the time stamp of the keystroke in milliseconds.

Comments

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 **KeyData** 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. For more information on extended ASCII codes, see Appendix C, “Code Pages.”

The **KbdPeek** function must be called twice to retrieve a double-byte character set (DBCS) code. The **fbStatus** field of the **KeyData** structure is set to 0x0080 only if the interim character flag is set. For more information about getting and setting the interim character flags, see the **KbdGetStatus** and **KbdSetStatus** functions.

Family API 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 keyboard (handle 0).
- The **fbStatus** field may be 0x0000 or 0x0001.
- 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 was already a character in the buffer, the **fbStatus** field specifies this and includes 0x40:

```
KBDKEYINFO kbciKeyInfo;
.
.
.
KbdPeek(&kbciKeyInfo, 0);
if (kbciKeyInfo.fbStatus & 0x40) {    /* if character received */
```

See Also

KbdCharIn

■ **USHORT KbdRegister**(*pszModuleName*, *pszEntryName*, *fFunctions*)
PSZ *pszModuleName*; pointer to module name
PSZ *pszEntryName*; pointer to entry-point name
ULONG *fFunctions*; function flags

The **KbdRegister** function registers a keyboard subsystem for the specified logical keyboard. The function temporarily replaces the one or more default keyboard functions, as specified by the *fFunctions* parameter,

with the functions in the module named by the *pszModuleName* parameter. Once **KbdRegister** replaces a function, the system passes any subsequent call to the replaced function to a function in the given module. If a function is not replaced, the system continues to call the default keyboard function.

The system passes a keyboard function to be the given module by preparing the stack and calling the function named the *pszEntryName* parameter. The entry-point function name must be exported by the specified module. The entry-point function must determine which function is being requested (by checking the function code on the stack), and then pass control to the appropriate function in the module. The function may then access any additional parameters placed on the stack by the original call.

Parameter	Description
<i>pszModuleName</i>	Points to a null-terminated string that specifies the name of the dynamic-link module that contains the replacement keyboard functions. The string must be a valid MS OS/2 filename.
<i>pszEntryName</i>	Points to a null-terminated string that specifies the dynamic-link entry-point name of the function that replaces the specified keyboard function(s). For a full description, see the following "Comments" section.
<i>fFunctions</i>	Specifies the keyboard function(s) to be replaced. It 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**.
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. The error value may be one of the following:

ERROR_KBD_INVALID_ASCII
The null-terminated string length is invalid.
ERROR_KBD_INVALID_MASK
The replacement mask is invalid.
ERROR_KBD_REGISTER
The **KbdRegister** function is not allowed.

Comments

Only one process in a screen group may use the **KbdRegister** function at any given time. That is, only one process can replace keyboard functions at any given time. The process can restore the default keyboard functions by calling the **KbdDeRegister** function. A process can replace keyboard 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 must have the following form:

```
VOID 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
<i>selDataSeg</i>	Specifies the data-segment selector of the process that calls the keyboard function.
<i>usReserved1</i>	Specifies a reserved value that must not be changed. It represents a return address for the system function that routes keyboard function calls.
<i>fFunction</i>	Specifies the function code of the function request. It 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.

ulReserved2 Specifies a reserved value that must not be changed. It represents the return address to the program that calls the specified keyboard function.

usParam1–usParam6 Specifies up to six unsigned values passed with the original keyboard function call. Not all requests include all six parameters since not all keyboard 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. If desired, the entry-point function may call a function within the same module to carry out the task. The 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 function.

If the function needs to access the keyboard, it must use the input-and-output control functions for the keyboard as described in Chapter 4, “Input-and-Output Control Functions.”

The **KbdRegister** function itself cannot be replaced.

See Also

KbdDeRegister, **KbdFlushBuffer**

■ **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 the system 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 if the default translation table for the logical keyboard is desired.

The **KbdSetCp** function also flushes the input buffer of the logical keyboard so that any keystrokes translated using the previous translation table are removed.

Parameter	Description
<i>usReserved</i>	Specifies a reserved value. It must be zero.

idCodePage Specifies the code-page identifier. If the identifier is 0x0000, the default translation table is used.

hkbd Identifies the logical keyboard device. The handle must have been previously opened 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, "Code Pages."

Example

This example calls **KbdSetCp** to change the keyboard subsystem so that it uses the U.S. multilingual code page when translating keystrokes for the default keyboard. The code page must be installed in the *config.sys* file or this function returns an error value:

```
KbdSetCp(OL,          /* reserved */
          850,         /* code-page identifier */
          0);          /* keyboard handle */
```

See Also

DosSetCp, **KbdGetCp**, **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. The system 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. A translation table remains in use until the process uses the **KbdSetCp** or **KbdSetCustXt** function to set another translation table, or uses the **KbdClose** function to close the logical keyboard.

Parameter	Description
<i>pusTransTbl</i>	Points to the translation table. The table has the size and format described in Appendix C, “Code Pages.”
<i>hkbd</i>	Identifies the logical keyboard that uses the new code page. The handle must have been previously opened 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

KbdSetCp, **KbdXlate**

- **USHORT KbdSetStatus**(*pkbstKbdInfo*, *hkbd*)
PKBDINFO *pkbstKbdInfo*; pointer to buffer with 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, turn-around character, interim character flags, and shift state.

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

Parameter	Description
<i>pkbstKbdInfo</i>	Points to the KBDINFO structure that contains the keyboard status. For a full description, see the following “Structures” section.
<i>hkbd</i>	Identifies the logical keyboard. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

ERROR_KBD_INVALID_ECHO_MASK
The echo on/off mode is invalid.

KbdSetStatus

ERROR_KBD_INVALID_INPUT_MASK
The raw/cooked mode is invalid.

ERROR_KBD_INVALID_LENGTH
The structure length is invalid.

Structures

The **KBDINFO** structure pointed to by the *pkbstKbdInfo* parameter has the following form:

```
typedef struct _KBDINFO {  
    USHORT cb;  
    USHORT fsMask;  
    USHORT chTurnAround;  
    USHORT fsInterim;  
    USHORT fsState;  
} KBDINFO;
```

Field	Description																						
cb	Specifies the length in bytes of the KBDINFO structure. The value of the cb field must be set to 0x000A before the function is called.																						
fsMask	Specifies the current keyboard modes. It can be any combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>No states set.</td></tr><tr><td>0x0001</td><td>Echo mode turned on.</td></tr><tr><td>0x0002</td><td>Echo mode turned off.</td></tr><tr><td>0x0004</td><td>Raw mode turned on.</td></tr><tr><td>0x0008</td><td>Cooked mode turned on.</td></tr><tr><td>0x0010</td><td>The fsState field is to be modified.</td></tr><tr><td>0x0020</td><td>The fsInterim field is to be modified.</td></tr><tr><td>0x0040</td><td>The chTurnAround field is to be modified.</td></tr><tr><td>0x0080</td><td>Two-byte turn-around character. If not given, the turn-around character is one byte.</td></tr><tr><td>0x0100</td><td>Shift report on.</td></tr></table> Echo mode is either turned on or off, not both. Only one input mode, raw or cooked, can be turned on at any given time.	Value	Meaning	0x0000	No states set.	0x0001	Echo mode turned on.	0x0002	Echo mode turned off.	0x0004	Raw mode turned on.	0x0008	Cooked mode turned on.	0x0010	The fsState field is to be modified.	0x0020	The fsInterim field is to be modified.	0x0040	The chTurnAround field is to be modified.	0x0080	Two-byte turn-around character. If not given, the turn-around character is one byte.	0x0100	Shift report on.
Value	Meaning																						
0x0000	No states set.																						
0x0001	Echo mode turned on.																						
0x0002	Echo mode turned off.																						
0x0004	Raw mode turned on.																						
0x0008	Cooked mode turned on.																						
0x0010	The fsState field is to be modified.																						
0x0020	The fsInterim field is to be modified.																						
0x0040	The chTurnAround field is to be modified.																						
0x0080	Two-byte turn-around character. If not given, the turn-around character is one byte.																						
0x0100	Shift report on.																						

- chTurnAround** Specifies the turn-around character. If the **fsMask** field value includes 0x0080, the character is two-bytes packed in the low- and high-order bytes of this field. Otherwise, the character is a single byte in the low-order byte.
- fsInterim** Specifies the interim character flags. If the **fsInterim** field is 0x0020, the program has requested character conversion. If it is 0x0080, the interim character flag is turned on.
- fsState** Specifies the state of the shift keys. It can be any combination of the following values:

Value	Meaning
0x0000	SHIFT key up.
0x0001	Right SHIFT key down.
0x0002	Left SHIFT key down.
0x0004	CONTROL key down.
0x0008	ALT key down.
0x0010	SCROLL LOCK key turned on.
0x0020	NUMLOCK key turned on.
0x0040	CAPSLOCK key turned on.
0x0080	INSERT key turned on.

Comments

In most cases, shift-state keys are pressed in combination with other keys to create a single character. In raw mode with shift report turned on, a shift-state key by itself creates a character that can be retrieved by the **KbdCharIn** or **KbdPeek** function.

Family API Restrictions

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

- Interim and turnaround characters are not supported.
- The raw input mode with echo mode on is not supported.
- The *hkbd* parameter is ignored.

Example

This example gets the current keyboard status, masks the cooked-mode bit, uses the OR operator to set the raw-mode bit, and calls the **KbdSetStatus** function to change the keyboard status to raw mode:

```
KBDINFO kbstKbdInfo;  
kbstKbdInfo.cb = sizeof(kbstKbdInfo);  
KbdGetStatus(&kbstKbdInfo, 0);      /* Gets the current status */  
kbstKbdInfo.fsMask =  
    (kbstKbdInfo.fsMask & 0x00F7)    /* Masks out cooked mode */  
    | 0x0004;                        /* OR in raw mode */  
KbdSetStatus(&kbstKbdInfo, 0);      /* Sets the new status */
```

See Also

KbdCharIn, **KbdGetStatus**, **KbdPeek**

- **USHORT KbdStringIn**(*pchBuffer*, *psibLength*, *fNoWait*, *hkbd*)
PCH *pchBuffer*; pointer to buffer for string
PSTRINGINBUF *psibLength*; pointer to buffer with max. string length
USHORT *fNoWait*; 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 *fNoWait* parameter, **KbdStringIn** continues to copy characters until it fills the buffer, retrieves the turn-around character, or finds that no more characters are available.

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

Parameter	Description
<i>pchBuffer</i>	Points to the buffer that receives the character string.
<i>psibLength</i>	Points to the STRINGINBUF structure that contains the length of the buffer that receives the string. For a full description, see the following “Structures” section.
<i>fNoWait</i>	Specifies whether to wait for the entire string to be read. If the <i>fNoWait</i> parameter is IO_WAIT , the function waits for all characters up to the next turn-around character or until the end of the buffer is read. If it is IO_NOWAIT , the function returns immediately with whatever characters are available.
<i>hkbd</i>	Identifies the logical keyboard device to read from. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

Structures

The **STRINGINBUF** structure pointed to by the *psibLength* parameter has the following form:

```
typedef struct _STRINGINBUF {
    USHORT cb;
    USHORT cchIn;
} STRINGINBUF;
```

Field	Description
cb	Specifies the length in bytes of the buffer. The maximum value for the cb field is 0x00FF bytes.
cchIn	Specifies the number of bytes read. The maximum value for the cchIn field is 0x00FF bytes.

Comments

The **KbdStringIn** function removes keystroke information from the input buffer of the specified logical keyboard as it copies a character. If echo and cooked modes are turned on, the function echoes characters on the screen as they are typed. If the keyboard is in cooked mode, the function retrieves characters for each key pressed, except shift keys, MS OS/2 control keys, and MS OS/2 editing keys. If the keyboard is in raw mode, the function retrieves characters 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. For more information on extended ASCII codes, see Appendix C, "Code Pages." In cooked mode, the function retrieves only complete extended codes; that is, if both bytes of the extended code do not fit in the buffer, neither byte is copied.

In cooked 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 **KbdStringInLength** structure is set to the length of the previous string before the function is called. If this field is zero, the line cannot be edited.

Family API 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 cooked mode, the function waits for the RETURN key to be pressed, and in raw 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,         /* wait/no wait              */
            0);              /* keyboard handle           */
VioWrtTTY("\n", 1, 0);       /* Sends a linefeed character */
VioWrtTTY(achBuf, kbsiBuf.cchIn, 0); /* Displays the string      */
```

See Also

DosRead, KbdCharIn, KbdGetStatus, KbdSetStatus

■ **USHORT KbdSynch(*fNoWait*)**
USHORT *fNoWait*; wait/no-wait flag

The **KbdSynch** function synchronizes access to the keyboard device driver.

This function is to be used by a subsystem, not by a program. The **KbdSynch** function cannot be replaced by using the **KbdRegister** function.

Parameter	Description
<i>fNoWait</i>	Specifies whether to wait for access to the keyboard router if access is not available. If the <i>fNoWait</i> parameter is IO_WAIT , the function waits for access to the keyboard router. If it is IO_NOWAIT , the function does not wait.

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 keyboard function is completed.

See Also**DosDevIOCtl**

■ **USHORT KbdXlate**(*pkbzlKeyStroke*, *hkbd*)
PKBDTRANSLATE *pkbzlKeyStroke*; pointer to buffer with keystroke
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.

Parameter	Description
<i>pkbzlKeyStroke</i>	Points to the KBDTRANSLATE structure that contains the scan code to be translated. It also receives the character value when the function returns. For a full description, see the following “Structures” section.
<i>hkbd</i>	Identifies the logical keyboard device. The handle must have been previously opened by using the KbdOpen function.

Return Value

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

Structures

The **KBDTRANSLATE** structure pointed to by the *pkbzlKeyStroke* parameter has the following form:

```
typedef struct _KBDTRANSLATE {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
    USHORT fsDD;
    USHORT fsXlate;
    USHORT fsShift;
    USHORT sZero;
} KBDTRANSLATE;
```

Field	Description												
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: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>Shift key received (valid only in raw mode when shift report is turned on).</td></tr> <tr> <td>0x0020</td><td>Conversion requested.</td></tr> <tr> <td>0x0040</td><td>Final character received.</td></tr> <tr> <td>0x0080</td><td>Interim character received.</td></tr> </table>	Value	Meaning	0x0001	Shift key received (valid only in raw mode when shift report is turned on).	0x0020	Conversion requested.	0x0040	Final character received.	0x0080	Interim character received.		
Value	Meaning												
0x0001	Shift key received (valid only in raw mode when shift report is turned on).												
0x0020	Conversion requested.												
0x0040	Final character received.												
0x0080	Interim character received.												
bNlsShift	Specifies a reserved value. It must be zero.												
fsState	Specifies the state of the shift keys. It can be one of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>SHIFT key up.</td></tr> <tr> <td>0x0001</td><td>Right SHIFT key down.</td></tr> <tr> <td>0x0002</td><td>Left SHIFT key down.</td></tr> <tr> <td>0x0004</td><td>Either CONTROL key down.</td></tr> <tr> <td>0x0008</td><td>Either ALT key down.</td></tr> </table>	Value	Meaning	0x0000	SHIFT key up.	0x0001	Right SHIFT key down.	0x0002	Left SHIFT key down.	0x0004	Either CONTROL key down.	0x0008	Either ALT key down.
Value	Meaning												
0x0000	SHIFT key up.												
0x0001	Right SHIFT key down.												
0x0002	Left SHIFT key down.												
0x0004	Either CONTROL key down.												
0x0008	Either ALT key down.												

0x0010	SCROLL LOCK key turned on.
0x0020	NUMLOCK key turned on.
0x0040	CAPSLOCK key turned on.
0x0080	INSERT key turned on.
0x0100	Left CONTROL key down.
0x0200	Left ALT key down.
0x0400	Right CONTROL key down.
0x0800	Right ALT key down.
0x1000	SCROLL LOCK key down.
0x2000	NUMLOCK key down.
0x4000	CAPSLOCK key down.
0x8000	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 the fsXlate 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 the fsShift field should not be touched unless a new translation is desired. This field is cleared at the completion of the translation.
sZero	Specifies a reserved value. It must be zero.

Comments

Accent-key combinations, double-byte characters, and extended ASCII characters may require several calls to the **KbdXlate** function to be translated.

See Also

DosMonReg, **KbdSetCustXt**

- **USHORT MouClose(*hmou*)**
HMOU *hmou*; mouse handle

The **MouClose** function closes the mouse device identified by the given handle. The function removes the mouse pointer from the screen only if the process is the last process in the screen group to have the mouse open.

Parameter	Description
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Comments

This function returns an error value if no mouse device is attached.

Example

This example calls the **MouClose** function to close an open mouse device handle:

```
HMOU hmou;  
MouOpen(OL, &hmou);  
.  
.  
.  
MouClose(hmou);
```

See Also

MouOpen

- **USHORT MouDeRegister(VOID)**

The **MouDeRegister** function restores the default mouse subsystem functions and releases any previously registered mouse subsystem. This function restores the default mouse subsystem for all processes in the current screen group.

This function has no parameters.

Return Value

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

ERROR_MOUSE_DEREGISTER
No alternate subsystem is registered.

Comments

Once a process registers a mouse subsystem, no other process in the screen group may register a mouse subsystem until the default mouse subsystem is restored. Only the process that registers a mouse subsystem may call the **MouDeRegister** function to restore the default mouse subsystem.

See Also

MouRegister

■ **USHORT MouDrawPtr(*hmou*)**
HMOU *hmou*; mouse handle

The **MouDrawPtr** function draws the mouse pointer 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 region of the screen in which the system will not display the pointer.

Parameter	Description
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Comments

The **MouDrawPtr** function does not actually draw the mouse pointer. Instead, it directs the system to call the pointer-draw function at each mouse interrupt. The pointer-draw function combines the AND and XOR masks of the pointer shape with the content of the screen at the current mouse location to create the pointer. If the pointer-draw function has been disabled (by using the **MouSetDevStatus** function), **MouDrawPtr** releases the current exclusion rectangle but no pointer is drawn.

Example

This example 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**

- **USHORT MouFlushQue(*hmou*)**
HMOU *hmou*; mouse handle

The **MouFlushQue** function flushes the mouse event queue; that is, it removes any existing mouse events from the queue.

Parameter	Description
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Example

This example calls the **MouFlushQue** function to flush the existing queue for the current mouse device:

```
HMOU hmou;
MouOpen(OL, &hmou);
.
.
.
MouFlushQue(hmou);
```

See Also

MouGetNumQueEl, **MouOpen**, **MouReadEventQue**

- **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 device.

Parameter	Description														
<i>pfsDevStatus</i>	Points to a variable that receives the device status. It can be any combination of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>Event queue busy with input/output (I/O).</td></tr> <tr> <td>0x0002</td><td>Block read in progress.</td></tr> <tr> <td>0x0004</td><td>Flush buffer in progress.</td></tr> <tr> <td>0x0008</td><td>Pointer-draw function disabled because of unsupported mode.</td></tr> <tr> <td>0x0100</td><td>Pointer-draw function is disabled.</td></tr> <tr> <td>0x0200</td><td>Mouse motion given in mickeys, not in pixels.</td></tr> </table>	Value	Meaning	0x0001	Event queue busy with input/output (I/O).	0x0002	Block read in progress.	0x0004	Flush buffer in progress.	0x0008	Pointer-draw function disabled because of unsupported mode.	0x0100	Pointer-draw function is disabled.	0x0200	Mouse motion given in mickeys, not in pixels.
Value	Meaning														
0x0001	Event queue busy with input/output (I/O).														
0x0002	Block read in progress.														
0x0004	Flush buffer in progress.														
0x0008	Pointer-draw function disabled because of unsupported mode.														
0x0100	Pointer-draw function is disabled.														
0x0200	Mouse motion given in mickeys, not in pixels.														
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.														

Return Value

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

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

Example

This example calls the **MouGetDevStatus** function to retrieve the status for the specified mouse device:

```
USHORT fsDevStatus;  
HMOU hmou;  
MouOpen(OL, &hmou);  
MouGetDevStatus(&fsDevStatus, hmou);
```

See Also

MouOpen, MouSetDevStatus

■ **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 device. The event mask specifies which user actions the system generates mouse events for. The system reports a user action by copying a mouse event to the event queue.

Parameter	Description
<i>pfsEvents</i>	Points to a variable that receives the event mask. It can be any combination of the following values:
Value	Meaning
0x0001	Mouse motion.
0x0002	Mouse motion with button-1-down events.
0x0004	Button-1-down events.
0x0008	Mouse motion with button-2-down events.
0x0010	Button-2-down events.

	0x0020	Mouse motion with button-3-down events.
	0x0040	Button-3-down events.
<i>hmou</i>		Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Comments

Button 1 is the left button on the mouse.

Example

This example opens the mouse device, 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(OL, &hmou);
MouGetEventMask(&fsEvents, hmou);
if(fsEvents > 0x001F)
    VioWrtTTY("Three buttons enabled\n\r", 23, 0);
```

See Also

MouOpen, MouReadEventQue, MouSetEventMask

■ **USHORT MouGetHotKey(*pfsButtons*, *hmou*)**
PUSHORT *pfsButtons*; pointer to buffer for button mask
HMOU *hmou*; mouse handle

The **MouGetHotKey** function retrieves a button mask for the specified mouse device. The button mask specifies which button or buttons represent the system hot key.

Parameter	Description
<i>pfsButtons</i>	Points to a variable that receives the button mask. It can be any combination of the following values:

	Value	Meaning
	0x0001	No hot key defined.
	MHK_BUTTON1	Button 1 is the hot key.
	MHK_BUTTON2	Button 2 is the hot key.
	MHK_BUTTON3	Button 3 is the hot key.
	If the value is 0x0001 or any combination of values including 0x0001, no system hot key is defined.	
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by the MouOpen function.	

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Comments

Button 1 is the left button on the mouse.

If more than one button is specified, the system hot key is a combination of the given keys; that is, the user must press the given buttons at the same time to generate a system hot key.

Example

This example opens the mouse device and calls the **MouGetHotKey** function to determine if the first mouse button is the hot key. The hot key is set by the system manager for the entire system:

```
HMOU hmou;
USHORT fsButtons;
MouOpen(OL, &hmou);
MouGetHotKey(&fsButtons, hmou);
if(fsButtons == MHK_BUTTON1)
    VioWrtTTY("Button 1 is the hot key\n\r", 27, 0);
```

See Also

MouOpen, MouSetHotKey

- **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.

Parameter	Description
<i>pusButtons</i>	Points to a variable that receives the number of buttons on the mouse.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE

Example

This example calls the **MouGetNumButtons** function to retrieve the number of buttons on the mouse device:

```
HMOU hmou;
USHORT usButtons;
MouOpen(OL, &hmou);
MouGetNumButtons(&usButtons, hmou);
if(usButtons == 2)
    VioWrTtTy("Your mouse has two buttons\n\r", 28, 0);
```

See Also

MouOpen

- **USHORT MouGetNumMickeyes**(*pusMickeyes*, *hmou*)
PUSHORT *pusMickeyes*; pointer to variable for mickeys per centimeter
HMOU *hmou*; mouse handle

The **MouGetNumMickeyes** function retrieves the number of mickeys that the specified mouse device travels for each centimeter of motion. A

MouGetNumMickeys

mickey is the smallest unit of motion a mouse device can measure. The number of mickeys per centimeter for a mouse device depends on the device and may also depend on the current setting of the device.

Parameter	Description
<i>pusMickeys</i>	Points to a variable that receives the number of mickeys per centimeter.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Example

This example calls the **MouGetNumMickeys** function to retrieve the current number of mickeys per centimeter:

```
HMOU hmou;  
USHORT usMickeys;  
MouGetNumMickeys(&usMickeys, hmou);
```

See Also

MouOpen

- **USHORT MouGetNumQueEl**(*pmouqiNumEvents*, *hmou*)
PMOQUEUEINFO *pmouqiNumEvents*; pointer to buffer for number of events
HMOU *hmou*; mouse handle

The **MouGetNumQueEl** function retrieves the number of events in the mouse event queue.

Parameter	Description
<i>pmouqiNumEvents</i>	Points to a MOUQUEUEINFO structure. For a full description, see the following “Structures” section.

hmou Identifies the mouse device. The handle must have been opened by using the **MouOpen** function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Structures

The **MOUQUEINFO** structure pointed to by the *pmouqiNumEvents* parameter has the following form:

```
typedef struct _MOUQUEINFO {
    USHORT cEvents;
    USHORT cmaxEvents;
} MOUQUEINFO;
```

Field	Description
cEvents	Specifies the current number of event-queue elements. It can be any value between zero and the maximum queue size.
cmaxEvents	Specifies the maximum queue size, that is, the maximum number of queue elements.

Example

This example opens the mouse device, draws the mouse pointer, and executes within an infinite **for** loop until there are no events in the queue:

```
HMOU hmou;
MOUEVENTINFO mouevEvent;
MOUQUEINFO mouqiNumEvents;
USHORT fWait = FALSE;
MouOpen(OL, &hmou); /* Opens the mouse device */
MouDrawPtr(hmou); /* Shows the mouse pointer */
for (;;) {
    MouGetNumQueEl(&mouqiNumEvents, hmou); /* Retrieves the queue */
    if (mouqiNumEvents.cEvents > 1) /* until the last queue */
        MouReadEventQue(&mouevEvent, &fWait, hmou);
    else
        break;
}
```

See Also

MouFlushQue, MouOpen, MouReadEventQue

- **USHORT** **MouGetPtrPos**(*pmouplPosition*, *hmou*)
PPTRLOC *pmouplPosition*; pointer to buffer for position
HMOU *hmou*; mouse handle

The **MouGetPtrPos** function retrieves the current position of the mouse device. This position is given in screen coordinates.

Parameter	Description
<i>pmouplPosition</i>	Points to the PTRLOC structure that receives the coordinates of the mouse. For a full description, see the following “Structures” section.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Structures

The **PTRLOC** structure pointed to by the *pmouplPosition* parameter has the following format:

```
typedef struct _PTRLOC {  
    USHORT row;  
    USHORT col;  
} PTRLOC;
```

Field	Description
row	Specifies the <i>x</i> -coordinate of the mouse.
col	Specifies the <i>y</i> -coordinate of the mouse.

The values of the **row** and **col** fields depend on the current video mode of the screen (as defined by the **VioSetMode** function). If the video mode is text mode, the units are given in character cells; if it is graphics mode, the units are given in pixels.

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 opens the mouse device and draws the mouse pointer. It 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 mouplPosition;
HMOU hmou;
BYTE bAttr = 0x72;      /* green character on white background */
MouOpen(OL, &hmou);
MouDrawPtr(hmou);        /* Displays the mouse */
VioWrtCharStrAtt("Place mouse here", 16, 0, 35, &bAttr, 0);
do
    MouGetPtrPos(&mouplPosition, hmou);
while (mouplPosition.row != 0 ||
      (mouplPosition.col < 35 || mouplPosition.col > 50));
```

See Also

MouOpen, MouSetPtrPos

- **USHORT MouGetPtrShape**(*pbBuffer*, *pmoupsInfo*, *hmou*)

PBYTE <i>pbBuffer</i> ;	pointer to buffer for shape masks
PPTRSHAPE <i>pmoupsInfo</i> ;	pointer to buffer for shape information
HMOU <i>hmou</i> ;	mouse handle

The **MouGetPtrShape** function retrieves the AND and XOR masks that define the shape of the pointer for the specified mouse device. It also retrieves information about the pointer such as the width and height of masks and the location of the hot spot.

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. 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 shape.

Parameter	Description
<i>pbBuffer</i>	Points to the buffer that receives the masks.
<i>pmoupsInfo</i>	Points to the PTRSHAPE structure that receives the pointer information. For a full description, see the following “Structures” section.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_INV_PARMS

The pointer-shape buffer size is invalid.

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

Structures

The **PTRSHAPE** structure pointed to by the *pmoupsInfo* parameter has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

Field	Description
cb	Specifies the length in bytes of the AND and XOR masks.
col	Specifies the width of each mask. In text mode, the width is given in character cells; in graphics mode, it is given in pixels. It must be greater than or equal to 1.
row	Specifies the height of each mask. In text mode, the width is given in character cells; in graphics mode, it is given in pixels. It 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. In text mode, the offset is given in character cells; in graphics mode, it is given in pixels.

rowHot Specifies the vertical offset from the upper-left corner of the pointer shape to the hot spot. In text mode, the offset is given in character cells; in graphics mode, it is given in pixels.

The **cb** field of the **PTRSHAPE** structure is always equal to **row** times **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.

Comments

The **cb** field of the **PTRSHAPE** structure must be set to the appropriate buffer size before the **MouGetPtrShape** function is called. 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.

For a description of AND and XOR mask formats, see Appendix B, “Devices.”

Example

This example opens the mouse device, draws the mouse pointer, and calls the **MouGetPtrShape** function to retrieve the shape of the mouse pointer:

```
PTRSHAPE mouspInfo;
BYTE abBuffer[4];
HMOU hmou;
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
mouspInfo.cb = sizeof(abBuffer);
MouGetPtrShape(abBuffer, &mouspInfo, hmou);
```

See Also

MouOpen, **MouSetPtrShape**

■ **USHORT MouGetScaleFact**(*pmouseFactors*, *hmou*)
PSCALEFACT *pmouseFactors*; pointer to buffer for scaling factors
HMOU *hmou*; mouse handle

The **MouGetScaleFact** function retrieves the horizontal and vertical scaling factors for the specified mouse device. The scaling factors define the number of mickeys the mouse must travel horizontally or vertically to cause the system to move the mouse pointer one screen unit.

Parameter	Description
<i>pmouseFactors</i>	Points to the SCALEFACT structure that receives the scaling factors. For a full description, see the following “Structures” section.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Structures

The **SCALEFACT** structure pointed to by the *pmouseFactors* parameter has the following form:

```
typedef struct _SCALEFACT {  
    USHORT rowScale;  
    USHORT colScale;  
} SCALEFACT;
```

Field	Description
rowScale	Specifies the vertical scaling factor, that is, the number of mickeys the mouse must move to change the vertical mouse position by one screen unit.
colScale	Specifies the horizontal scaling factor, that is, the number of mickeys the mouse must move to change the horizontal mouse position by one screen unit.

The **rowScale** and **colScale** fields of the **SCALEFACT** structure specify mickeys and will always be in the range 1 to 32,767. The screen units may be character cells or pixels, depending on the current video mode.

Example

This example calls the **MouGetScaleFact** function to retrieve the row-and-column coordinate scaling factors:

```
SCALEFACT mouseFactors;  
HMOU hmou;  
MouGetScaleFact(&mouseFactors, hmou);
```

See Also

MouGetNumMickeys, MouOpen, MouSetScaleFact

■ USHORT MouInitReal(*pszDriverName*)

PSZ *pszDriverName*; pointer to pointer-draw driver name

The **MouInitReal** function initializes the real-mode mouse device driver. It loads and initializes the pointer-draw driver named by the *pszDriverName* parameter. The pointer-draw device-driver name must be specified by a **device** command in the *config.sys* file.

This function is intended to be used only by the session manager.

Parameter	Description
<i>pszDriverName</i>	Points to a null-terminated string that specifies the name of the pointer-draw driver. The name must be a valid MS OS/2 filename. The default pointer-draw function can be initialized by setting the <i>pszDriverName</i> parameter to zero.

Return Value

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

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Comments

The mouse 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. The **MouInitReal** function is like **MouOpen** for protected-mode programs in that it initializes the pointer-draw function but it does not provide a mouse handle. In the real-mode interface, no handle is required.

In many cases, the pointer-draw driver for protected mode also contains a pointer-draw function for real mode.

See Also

MouOpen

- **USHORT MouOpen**(*pszDriverName*, *phmou*)
PSZ *pszDriverName*; pointer to pointer-draw driver name
PHMOU *phmou*; pointer to variable for mouse handle

The **MouOpen** function opens the mouse device for the current screen group and creates a handle that can be used in subsequent mouse functions to display the mouse pointer, retrieve the current mouse location, and change the operation of the mouse.

The **MouOpen** function opens the mouse device for the current screen group only. Any number of processes may open the mouse, but all processes in the screen group share the mouse. For example, if one process changes the mouse pointer color, it changes the pointer color for all other processes in the same screen group.

When the mouse device is opened for the first time, **MouOpen** does not immediately display the mouse pointer. The **MouDrawPtr** function must be called to show the pointer. A pointer-draw driver is required to actually draw the pointer. If the pointer-draw driver named by the *pszDriverName* parameter does not exist or cannot be opened, the pointer will not be drawn. If *pszDriverName* is set to zero, the default pointer-draw driver is used; that is, the driver specified in a **device** command in the *config.sys* file is used.

Parameter	Description
<i>pszDriverName</i>	Points to a null-terminated string that specifies the name of the mouse-pointer-draw driver. The name must be a valid MS OS/2 filename. If zero is given, the default pointer-draw driver is used.
<i>phmou</i>	Points to a variable that receives the handle that identifies the mouse device.

Return Value

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

- ERROR_MOUSE_INV_MODULE**
The pointer-shape draw routine was not found.
- ERROR_MOUSE_NO_DEVICE**
No mouse device is attached.

Example

This example calls the **MouOpen** function to open a handle for use by the current screen group:

```
HMOU hmouse;
MouOpen(OL, &hmouse);
```

See Also

MouClose

■ **USHORT MouReadEventQue**(*pmousevEvent*, *pfWait*, *hmouse*)
PMOUSEEVENTINFO *pmousevEvent*; pointer to buffer for mouse event
PUSHORT *pfWait*; wait/no-wait flag
HMOU *hmouse*; mouse handle

The **MouReadEventQue** function retrieves a mouse event from the event queue of the specified mouse device. The event queue is a first-in, first-out buffer to which the system copies mouse events. A mouse event is a structure that specifies the user action that generated the event, as well as the location of the mouse and system time when the event occurred.

The system copies a mouse event to the event queue whenever the user moves the mouse, presses a mouse button, or releases a mouse button. The mouse event can specify a single action or a combination of actions, such as mouse motion with a button down. The system 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.

Parameter	Description
<i>pmousevEvent</i>	Points to the MOUSEEVENTINFO structure that receives the mouse event. For a full description, see the following “Structures” section.
<i>pfWait</i>	Points to a variable that specifies whether the function waits for an event. If the <i>pfWait</i> parameter is TRUE and the queue is empty, the function fills the MOUSEEVENTINFO structure with zeros and returns immediately. If it is FALSE, the function waits for a mouse event if none is available.
<i>hmouse</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_INV_PARMS

The *pfWait* parameter is invalid.

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

NO_ERROR_MOUSE_NO_DATA

Structures

The **MOUEVENTINFO** structure pointed to by the *pmousevEvent* parameter has the following form:

```
typedef struct _MOUEVENTINFO {  
    USHORT fs;  
    ULONG Time;  
    USHORT row;  
    USHORT col;  
} MOUEVENTINFO;
```

Field	Description																		
fs	Specifies the action that generated the mouse event. It can be any combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>No buttons down.</td></tr><tr><td>0x0001</td><td>Mouse moved with no buttons down.</td></tr><tr><td>0x0002</td><td>Mouse moved with button 1 down.</td></tr><tr><td>0x0004</td><td>Button 1 down.</td></tr><tr><td>0x0008</td><td>Mouse moved with button 2 down.</td></tr><tr><td>0x0010</td><td>Button 2 down.</td></tr><tr><td>0x0020</td><td>Mouse moved with button 3 down.</td></tr><tr><td>0x0040</td><td>Button 3 down.</td></tr></table>	Value	Meaning	0x0000	No buttons down.	0x0001	Mouse moved with no buttons down.	0x0002	Mouse moved with button 1 down.	0x0004	Button 1 down.	0x0008	Mouse moved with button 2 down.	0x0010	Button 2 down.	0x0020	Mouse moved with button 3 down.	0x0040	Button 3 down.
Value	Meaning																		
0x0000	No buttons down.																		
0x0001	Mouse moved with no buttons down.																		
0x0002	Mouse moved with button 1 down.																		
0x0004	Button 1 down.																		
0x0008	Mouse moved with button 2 down.																		
0x0010	Button 2 down.																		
0x0020	Mouse moved with button 3 down.																		
0x0040	Button 3 down.																		
Time	Specifies the time of day in milliseconds.																		
row	Specifies the <i>x</i> -coordinate of the mouse.																		
col	Specifies the <i>y</i> -coordinate of the mouse.																		

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 recent **MouSetDevStatus** function. Values may be absolute or relative. Units may be mickeys, character cells, or pixels.

Although a specific user action may not generate a mouse event, the **fs** field of the **MOUEVENTINFO** structure may include information about such an 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.

Example

This example opens the mouse device, draws the mouse pointer, 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 the mouse data */
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, MouSetEventMask

■ **USHORT MouRegister**(*pszModuleName, pszEntryName, flFunctions*)

PSZ <i>pszModuleName</i> ;	pointer to module name
PSZ <i>pszEntryName</i> ;	pointer to entry name
ULONG <i>flFunctions</i> ;	function flags

The **MouRegister** function registers a mouse subsystem for the specified mouse device. The function temporarily replaces the one (or more) default mouse functions, as specified by the *flFunctions* parameter, with the functions in the module named by the *pszModuleName* parameter. Once **MouRegister** replaces a function, the system passes any subsequent calls to the replaced function to a function in the given module. If a function is not replaced, the system continues to call the default mouse function.

Parameter	Description
<i>pszModuleName</i>	Points to a null-terminated string that specifies the name of the dynamic-link module that contains the replacement mouse functions.
<i>pszEntryName</i>	Points to a null-terminated string that specifies the dynamic-link entry-point name of the function that replaces the specified mouse function. For a full description, see the following “Comments” section.
<i>flFunctions</i>	Specifies the mouse function to be replaced. It can be any combination of the following values:
Value	Meaning
0x00000001	Replace MouGetNumButtons .
0x00000002	Replace MouGetNumMickey s.
0x00000004	Replace MouGetDevStatus .
0x00000008	Replace MouGetNumQueEl .
0x00000010	Replace MouReadEventQue .
0x00000020	Replace MouGetScaleFact .
0x00000040	Replace MouGetEventMask .
0x00000080	Replace MouSetScaleFact .
0x00000100	Replace MouSetEventMask .
0x00000200	Replace MouGetHotKey .
0x00000400	Replace MouSetHotKey .
0x00000800	Replace MouOpen .
0x00001000	Replace MouClose .
0x00002000	Replace MouGetPtrShape .
0x00004000	Replace MouSetPtrShape .
0x00008000	Replace MouDrawPtr .
0x00010000	Replace MouRemovePtr .
0x00020000	Replace MouGetPtrPos .
0x00040000	Replace MouSetPtrPos .
0x00080000	Replace MouInitReal .
0x00100000	Replace MouSetDevStatus .

Return Value

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

ERROR_MOUSE_INVALID_ASCII

The string length is invalid.

ERROR_MOUSE_INVALID_MASK

The replacement mask is invalid.

ERROR_MOUSE_REGISTER

The **MouRegister** function is not allowed.

Comments

The system passes a mouse function to the given module by preparing the stack and calling the function named by the *pszEntryName* parameter. The entry-point function name must be exported by the specified module. The entry-point function must determine which function is being requested (by checking the function code on the stack), then it passes control to the appropriate function in the module. The 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 **MouRegister** function at any given time. That is, only one process at a time can replace mouse functions. The process can restore the default mouse functions by calling the **MouDeRegister** function. A process can replace a mouse function any number of times, but only by first restoring the default functions and then re-registering the new functions.

The entry-point function for the replacement mouse functions must have the following form:

```
VOID 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																																												
<i>usReserved1</i>	Specifies a reserved value that must not be changed. It actually represents a return address for the system function that routes mouse function calls.																																												
<i>usFunction</i>	Specifies the function code that identifies the function request. It can be one of the following values:																																												
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>MouGetNumButtons called.</td></tr> <tr> <td>0x0001</td><td>MouGetNumMickeys called.</td></tr> <tr> <td>0x0002</td><td>MouGetDevStatus called.</td></tr> <tr> <td>0x0003</td><td>MouGetNumQueEl called.</td></tr> <tr> <td>0x0004</td><td>MouReadEventQue called.</td></tr> <tr> <td>0x0005</td><td>MouGetScaleFact called.</td></tr> <tr> <td>0x0006</td><td>MouGetEventMask called.</td></tr> <tr> <td>0x0007</td><td>MouSetScaleFact called.</td></tr> <tr> <td>0x0008</td><td>MouSetEventMask called.</td></tr> <tr> <td>0x0009</td><td>MouGetHotKey called.</td></tr> <tr> <td>0x000A</td><td>MouSetHotKey called.</td></tr> <tr> <td>0x000B</td><td>MouOpen called.</td></tr> <tr> <td>0x000C</td><td>MouClose called.</td></tr> <tr> <td>0x000D</td><td>MouGetPtrShape called.</td></tr> <tr> <td>0x000E</td><td>MouSetPtrShape called.</td></tr> <tr> <td>0x000F</td><td>MouDrawPtr called.</td></tr> <tr> <td>0x0010</td><td>MouRemovePtr called.</td></tr> <tr> <td>0x0011</td><td>MouGetPtrPos called.</td></tr> <tr> <td>0x0012</td><td>MouSetPtrPos called.</td></tr> <tr> <td>0x0013</td><td>MouInitReal called.</td></tr> <tr> <td>0x0014</td><td>MouSetDevStatus called.</td></tr> </table>	Value	Meaning	0x0000	MouGetNumButtons called.	0x0001	MouGetNumMickey s 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	MouSetDevStatus called.
Value	Meaning																																												
0x0000	MouGetNumButtons called.																																												
0x0001	MouGetNumMickey s 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	MouSetDevStatus called.																																												
<i>ulReserved2</i>	Specifies a reserved value that must not be changed. It represents the return address to the program that calls the specified mouse function.																																												

usParam1–usParam5

Specifies up to five values passed with the original mouse function call. Not all requests include all five parameters since not all mouse functions use five parameters. The actual number and type of parameters depends on the specific function.

The entry-point function should determine which function is requested, and then carry out an appropriate action using the passed parameters. If desired, the entry-point function may call a replacement function within the given module to carry out the task. In any case, the entry-point or replacement function must also leave the stack in the same state it received it. 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 function.

In general, if the replacement function needs to access the mouse, it must use the input-and-output control functions for the mouse. These are described in Chapter 4, “Input-and-Output Control Functions.”

The **MouRegister** function itself cannot be replaced.

See Also

MouDeRegister

■ **USHORT MouRemovePtr**(*pmourtRect*, *hmou*)
PNOPTRRECT *pmourtRect*; pointer to buffer 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. The function actually creates an exclusion rectangle in which the mouse pointer disappears whenever it moves into the rectangle. The pointer is still present and can be moved, but it is not displayed until it is moved out of the exclusion rectangle.

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 removes the previous rectangle. The **MouDrawPtr** function removes the exclusion rectangle completely.

Parameter	Description
<i>pmourtRect</i>	Points to the NOPTRRECT structure that contains the upper-left corner, width, and height of the exclusion rectangle. For a full description, see the following “Structures” section.

hmouse Identifies the mouse device. The handle must have been opened by using the **MouOpen** function.

Return Value

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

ERROR_MOUSE_INV_PARMS

The coordinates are invalid or the size is invalid.

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

Structures

The **NOPTRRECT** structure pointed to by the *pmouseRect* parameter has the following form:

```
typedef struct _NOPTRRECT {
    USHORT row;
    USHORT col;
    USHORT cRow;
    USHORT cCol;
} NOPTRRECT;
```

Field	Description
row	Specifies the <i>x</i> -coordinate of the upper-left corner.
col	Specifies the <i>y</i> -coordinate of the upper-left corner.
cRow	Specifies the <i>x</i> -coordinate of the lower-right corner.
cCol	Specifies the <i>y</i> -coordinate of the lower-right corner.

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 pixels. The fields must not exceed the minimum and maximum coordinate values for screen height and width.

Comments

If the pointer is not currently displayed and it is outside the new exclusion rectangle, **MouRemovePtr** draws the pointer.

Example

This example opens the mouse device and draws the mouse pointer. It defines an area in the center of the screen and calls the **MouRemovePtr**

function to notify the mouse driver that this area is for the exclusive use of the process. When the mouse is moved into this area, it disappears:

```
NOPTRRECT mourtRect;
HMOU hmou;
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
mourtRect.row = 6;
mourtRect.col = 30;
mourtRect.cRow = 18;
mourtRect.cCol;
MouRemovePtr(&mourtRect, hmou);
```

See Also

MouDrawPtr, MouOpen, MouSetPtrShape

- **USHORT MouSetDevStatus**(*pfsDevStatus*, *hmou*)
PUSHORT *pfsDevStatus*; pointer to buffer with status
HMOU *hmou*; mouse handle

The **MouSetDevStatus** function sets the device fbStatus for the specified mouse device. The device status enables or disables the pointer-draw function and defines whether the mouse position is reported as mickeys or pixels.

Parameter	Description						
<i>pfsDevStatus</i>	Points to a variable that contains the device status to be set. It can be any combination of the following values:						
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0100</td><td>Disable the default pointer-draw function. If this value is not given, the function enables the pointer-draw function.</td></tr> <tr> <td>0x0200</td><td>Report mouse motion in mickeys; that is, the system reports motion as a number of mickeys moved from the last-reported position. If the value is not given, the system reports mouse motion in screen units (character cells or pixels) relative to the upper-left corner of the screen.</td></tr> </table>	Value	Meaning	0x0100	Disable the default pointer-draw function. If this value is not given, the function enables the pointer-draw function.	0x0200	Report mouse motion in mickeys; that is, the system reports motion as a number of mickeys moved from the last-reported position. If the value is not given, the system reports mouse motion in screen units (character cells or pixels) relative to the upper-left corner of the screen.
Value	Meaning						
0x0100	Disable the default pointer-draw function. If this value is not given, the function enables the pointer-draw function.						
0x0200	Report mouse motion in mickeys; that is, the system reports motion as a number of mickeys moved from the last-reported position. If the value is not given, the system reports mouse motion in screen units (character cells or pixels) relative to the upper-left corner of the screen.						
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.						

Return Value

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

ERROR_MOUSE_INV_PARMS

The parameters are invalid or out of range.

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

Comments

The **MouSetDevStatus** function enables or disables the pointer-draw function. When the pointer-draw function 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 function 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 device 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 pixels. Screen coordinates increase on the x -axis from left to right. They increase on the y -axis from top to bottom. If the device is set to report relative positions, the x - and y -coordinates given for a mouse position are given in mickeys and are relative to the last-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 device moves left; otherwise, it is positive. The y -coordinate is negative when the mouse device moves up; otherwise, it is positive.

Example

This example sets the device status to return the mouse movement information in terms of mickeys, not pixels. This allows the process to get mouse information in terms of relative movement rather than in terms of absolute pixel position:

```
USHORT fsDevStatus = 0x0200;          /* Returns mickeys */
HMOU hmou;
MouOpen(OL, &hmou);
MouSetDevStatus(&fsDevStatus, hmou);
```


See Also

MouGetDevStatus, MouOpen

■ **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 device. The event mask defines which user actions generate mouse events (movement, pressing a button, or releasing a button).

The **MouSetEventMask** function enables or disables specific user actions. When an action is enabled, the system 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.

Parameter	Description																
<i>pfsEvents</i>	Points to a variable that contains the event mask. The variable can be any combination of the following values:																
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>Enable mouse motion with no-buttons-down event.</td></tr> <tr> <td>0x0002</td><td>Enable mouse motion with button-1-down event.</td></tr> <tr> <td>0x0004</td><td>Enable button-1-down event.</td></tr> <tr> <td>0x0008</td><td>Enable mouse motion with button-2-down event.</td></tr> <tr> <td>0x0010</td><td>Enable button-2-down event.</td></tr> <tr> <td>0x0020</td><td>Enable mouse motion with button-3-down event.</td></tr> <tr> <td>0x0040</td><td>Enable button-3-down event.</td></tr> </table>	Value	Meaning	0x0001	Enable mouse motion with no-buttons-down event.	0x0002	Enable mouse motion with button-1-down event.	0x0004	Enable button-1-down event.	0x0008	Enable mouse motion with button-2-down event.	0x0010	Enable button-2-down event.	0x0020	Enable mouse motion with button-3-down event.	0x0040	Enable button-3-down event.
Value	Meaning																
0x0001	Enable mouse motion with no-buttons-down event.																
0x0002	Enable mouse motion with button-1-down event.																
0x0004	Enable button-1-down event.																
0x0008	Enable mouse motion with button-2-down event.																
0x0010	Enable button-2-down event.																
0x0020	Enable mouse motion with button-3-down event.																
0x0040	Enable button-3-down event.																
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.																

Return Value

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

ERROR_MOUSE_INV_PARMS
The input event mask is invalid.

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

Comments

Button 1 is the left button on the mouse.

Example

This example 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;  
fsEvents = 0x0007; /* Detects motion and button 1 */  
MouSetEventMask(&fsEvents, hmou);
```

See Also

MouGetEventMask, **MouOpen**, **MouReadEventQue**

- **USHORT** **MouSetHotKey**(*pfsButtons*, *hmou*)
PUSHORT *pfsButtons*; pointer to buffer with button mask
HMOU *hmou*; mouse handle

The **MouSetHotKey** function sets the system hot key for the specified mouse device. The system hot key applies to all screen groups; only one process in the system can call this function. It defines the hot key for both real and protected modes. This function is intended to be used by the session manager only.

Parameter	Description
<i>pfsButtons</i>	Points to a variable that contains the system hot-key mask. The variable can be any combination of the following values:
Value	Meaning
0x0001	No hot key.
MHK_BUTTON1	Button 1 is the hot key.
MHK_BUTTON2	Button 2 is the hot key.

MHK_BUTTON3

Button 3 is the hot key.

If the value is 0x0001 or any combination of values including 0x0001, no system hot key is defined.

hmouse

Identifies the mouse device. The handle must have been opened by using the **MouOpen** function.

Return Value

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

ERROR_MOUSE_CANT_RESET

Another process has defined the hot key.

ERROR_MOUSE_INV_PARMS

The input hot-key mask is invalid.

ERROR_MOUSE_NO_DEVICE

No mouse device is attached.

Comments

Button 1 is the left button on the mouse.

The mouse device driver saves the process identifier of the process that sets the hot key. Only the process with the saved process identifier may reset the hot key. The process that sets the button sequence must release it before another process may set it.

When a system hot key is defined, the system captures the press event and the release event of the hot key. That is, no corresponding mouse event is copied to the event queue. Although protected-mode monitors will recognize the event, the event will not be passed on to the event queue.

See Also

MouGetHotKey, **MouOpen**

■ **USHORT MouSetPtrPos**(*pmousePosition*, *hmouse*)
PPTRLOC *pmousePosition*; pointer to buffer with new position
HMOU *hmouse*; mouse handle

The **MouSetPtrPos** function sets the current mouse position to the position pointed to by the *pmousePosition* 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.

Parameter	Description
<i>pmouplPosition</i>	Points to the PTRLOC structure that contains the new mouse position. For a full description, see the following “Structures” section.
<i>hmouse</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_INV_PARMS
The coordinates are invalid.

ERROR_MOUSE_NO_DEVICE
No mouse device is attached.

Structures

The **PTRLOC** structure pointed to by the *pmouplPosition* parameter has the following form:

```
typedef struct _PTRLOC {  
    USHORT row;  
    USHORT col;  
} PTRLOC;
```

Field	Description
row	Specifies the <i>x</i> -coordinate of the mouse.
col	Specifies the <i>y</i> -coordinate of the mouse.

The **row** and **col** fields depend on the current video mode of the screen (as defined by the **VioSetMode** function). In text mode, the units are given in character cells; in graphics mode, the units are given in pixels.

Comments

The system 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 opens the mouse device and calls the **MouSetPtrPos** function to initialize the mouse pointer in the upper-left corner of the screen. It calls the **MouDrawPtr** function to actually display the mouse pointer:

```

PTRLOC mouplPosition;
HMOU hmou;
MouOpen(OL, &hmou);
mouplPosition.row = 0;          /* row zero          */
mouplPosition.col = 0;          /* column zero       */
MouSetPtrPos(&mouplPosition, hmou); /* Sets mouse position */
MouDrawPtr(hmou);              /* Displays the mouse */

```

See Also

MouDrawPtr, MouGetPtrPos, MouOpen, MouRemovePtr

- **USHORT MouSetPtrShape(*pbBuffer*, *pmoupsInfo*, *hmou*)**
PBYTE *pbBuffer*; pointer to buffer with shape masks
PPTRSHAPE *pmoupsInfo*; pointer to buffer 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 device. It also sets information about the pointer, such as the width and height of masks and the location of the hot spot.

Parameter	Description
<i>pbBuffer</i>	Points to the buffer that contains the new masks.
<i>pmoupsInfo</i>	Points to the PTRSHAPE structure that contains the new pointer information. For a full description, see the following “Structures” section.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

- ERROR_MOUSE_INV_PARMS**
The pointer-shape size or definition is invalid.
- ERROR_MOUSE_NO_DEVICE**
No mouse device is attached.

Structures

The **PTRSHAPE** structure pointed to by the *pmoupsInfo* parameter has the following form:

```
typedef struct _PTRSHAPE {  
    USHORT cb;  
    USHORT col;  
    USHORT row;  
    USHORT colHot;  
    USHORT rowHot;  
} PTRSHAPE;
```

Field	Description
cb	Specifies the length in bytes of the AND and XOR masks.
col	Specifies the width of each mask. In text mode, the width is given in character cells. In graphics mode, the width is given in pixels. The width must be greater than or equal to one.
row	Specifies the height of each mask. In text mode, the height is given in character cells. In graphics mode, the height is given in pixels. The height must be greater than or equal to one.
colHot	Specifies the horizontal offset from the upper-left corner of the pointer shape to the hot spot. In text mode, the offset is given in character cells; in graphics mode, it is given in pixels.
rowHot	Specifies the vertical offset from the upper-left corner of the pointer shape to the hot spot. In text mode, the offset is given in character cells; in graphics mode, it is given in pixels.

The **cb** field must be equal to **row** times **col**. If the current video mode requires multiple bit planes, the **row** and **col** fields must specify the width and height of the first plane only, and the buffer pointed to by the *pbBuffer* parameter must contain all bit planes.

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 display device driver used to support the display device. In text mode, the system 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. The current pointer shape in effect for the screen group may be determined by using the **MouGetPtrShape** function. For more information about pointer-shape formats, see Appendix B, “Devices.”

See Also

MouDrawPtr, **MouGetPtrShape**, **MouOpen**, **MouRemovePtr**

■ **USHORT MouSetScaleFact**(*pmouseFactors*, *hmou*)
PSCALEFACT *pmouseFactors*; pointer to buffer with new factors
HMOU *hmou*; mouse handle

The **MouSetScaleFact** function sets the horizontal and vertical scaling factors for the specified mouse device. The scaling factors define the number of mickeys the mouse must travel horizontally or vertically to cause the system to move the mouse pointer one screen unit.

Parameter	Description
<i>pmouseFactors</i>	Points to the SCALEFACT structure that contains the scaling factors. For a full description, see the following “Structures” section.
<i>hmou</i>	Identifies the mouse device. The handle must have been opened by using the MouOpen function.

Return Value

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

ERROR_MOUSE_NO_DEVICE
 No mouse device is attached.

Structures

The **SCALEFACT** structure pointed to by the *pmouseFactors* parameter has the following form:

```
typedef struct _SCALEFACT {  
    USHORT rowScale;  
    USHORT colScale;  
} SCALEFACT;
```

Field	Description
rowScale	Specifies the vertical-scaling factor; that is, the number of mickeys the mouse must move to change the vertical mouse position by one screen unit.
colScale	Specifies the horizontal-scaling factor; that is, the number of mickeys the mouse must move to change the horizontal mouse position by one screen unit.

The **rowScale** and **colScale** fields must specify mickeys and be a value from 1 to 32,767. The screen units may be character cells or pixels, depending on the current video mode.

Example

This example opens the mouse device, draws the mouse pointer, 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 the mouse must be moved twice as far to move the pointer on the screen:

```
SCALEFACT mouseFactors;  
HMOU hmou;  
MouOpen(OL, &hmou);  
MouDrawPtr(hmou);  
MouGetScaleFact(&mouseFactors, hmou);  
mouseFactors.rowScale *= 2;  
mouseFactors.colScale *= 2;  
MouSetScaleFact(&mouseFactors, hmou);
```

See Also

MouGetScaleFact, **MouOpen**

■ **USHORT MouSynch(*fWait*)**
USHORT *fWait*; wait/no-wait flag

The **MouSynch** function synchronizes access to the mouse device. This function is intended to be used by a mouse subsystem to prevent more than one process from accessing the mouse device at any time.

Parameter	Description
<i>fWait</i>	Specifies whether or not to wait if the mouse device driver is currently busy. If the <i>fWait</i> parameter is FALSE, the function returns control immediately without waiting for the device to become free. If it is TRUE, the function waits until the mouse device 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 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

■ USHORT VioDeRegister(VOID)

The **VioDeRegister** function restores the default video subsystem functions and releases any previously registered video subsystem. The function restores the default video subsystem for all processes in the current screen group.

This function has no parameters.

Return Value

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

ERROR_VIO_DEREGISTER

The **VioDeRegister** function is not allowed.

Comments

Once a process registers a video subsystem, no other process in the screen group may register a video subsystem until the default subsystem is restored. Only the process registering a video subsystem may call the **VioDeRegister** function to restore the default video subsystem.

See Also

VioRegister

■ USHORT VioEndPopUp(*hvio*)

HVIO *hvio*; video handle

The **VioEndPopUp** function closes a pop-up screen and restores the physical screen buffer to its previous contents. Only the process that opened the pop-up screen may close it.

Parameter	Description
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_NO_POPUP

Comments

In some cases, **VioEndPopUp** may not completely restore the screen to its previous state. For example, programs that modify the video registers or use graphics modes may need to restore the state of the registers as the pop-up window is being closed. A program can request to be informed of the change in video mode by using the **VioModeWait** function. Whenever a process has a pending request, the system notifies the process of a mode change when the pop-up is closed.

Example

This example creates a pop-up screen, displays a message, waits three seconds, and then calls **VioEndPopUp** to end the pop-up screen:

```
USHORT fWait = VP_WAIT;
VioPopUp(&fWait, 0);          /* Creates a pop-up screen */
VioWrtTTY("This is a VIO pop-up\n\r", 22, 0);
DosSleep(3000L);             /* Waits 3 seconds */
VioEndPopUp(0);              /* Ends the pop-up screen */
```

See Also

VioPopUp

- **USHORT VioGetAnsi**(*pfAnsi*, *hvio*)
PUSHORT *pfAnsi*; pointer to variable for ANSI state
HVIO *hvio*; video handle

The **VioGetAnsi** function retrieves the state of the ANSI flag. The ANSI flag specifies whether the **VioWrtTTY** function processes ANSI escape sequences.

Parameter	Description
<i>pfAnsi</i>	Points to a variable that receives the ANSI flag value. If it is ANSI_OFF , the ANSI state is off. If it is ANSI_ON , the ANSI state is on.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

Example

This example calls **VioGetAnsi** and displays a message if the ANSI flag state is turned on:

```
USHORT fAnsi;  
VioGetAnsi(&fAnsi, 0);  
if (fAnsi == ANSI_ON)  
    VioWrtTTY("ANSI is on\n\r", 12, 0);
```

See Also

VioSetAnsi

■ **USHORT VioGetBuf**(*pulVB*, *pcbVB*, *hvio*)
PULONG *pulVB*; pointer to buffer for LVB address
PUSHORT *pcbVB*; pointer to buffer for LVB length
HVIO *hvio*; video handle

The **VioGetBuf** function retrieves the address of the logical video buffer. The logical video buffer represents the current state of the process' text output to the screen and is identical in content and format to the physical screen buffer when the process is the foreground process. The logical video buffer is available for text mode screens only.

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

Parameter	Description
<i>pulVB</i>	Points to a variable that receives the address of the logical video buffer.
<i>pcbVB</i>	Points to a variable that specifies the length in bytes of the logical video buffer.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

Comments

A program may access and modify the contents of the logical video buffer at any time. Changes made to the logical video buffer do not affect the physical screen until the process switches from background to foreground or, if the process is already in the foreground, until the process calls the **VioShowBuf** function. The **VioGetMode** function can be used to determine the dimensions of the buffer.

Example

This example calls **VioGetBuf** to get the logical video buffer. It sets the attributes in the buffer for foreground blinking by setting the high bit with the OR operator, then it calls **VioShowBuf** to display the 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 the buffer */
```

See Also

VioGetMode, VioGetPhysBuf, VioShowBuf

- **USHORT VioGetConfig**(*usReserved*, *pvioinConfig*, *hvio*)

USHORT <i>usReserved</i> ;	Must be zero
PVIOCONFIGINFO <i>pvioinConfig</i> ;	pointer to buffer for configuration
HVIO <i>hvio</i> ;	video handle

The **VioGetConfig** function retrieves the video display configuration. The video display configuration defines the display adapter type, the monitor type, and the amount of video memory available.

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

Parameter	Description
<i>usReserved</i>	Specifies a reserved value. It must be zero.
<i>pvioinConfig</i>	Points to a VIOCONFIGINFO structure that receives the display configuration for the primary display adapter. For a full description, see the following "Structures" section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_INVALID_LENGTH**
The configuration-block length is invalid.
- ERROR_VIO_INVALID_PARMS**
The reserved word is not zero.

Structures

The **VIOCONFIGINFO** structure pointed to by the *pvioinConfig* parameter has the following format:

```
typedef struct _VIOCONFIGINFO {  
    USHORT cb;  
    USHORT adapter;  
    USHORT display;  
    ULONG cbMemory;  
} VIOCONFIGINFO;
```

Field	Description								
cb	Specifies the length of the structure in bytes. This field must be set to 0x000A before calling the VioGetConfig function.								
adapter	Specifies the display-adapter type with one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Monochrome/printer adapter</td></tr><tr><td>0x0001</td><td>Color graphics adapter</td></tr><tr><td>0x0002</td><td>Enhanced graphics adapter</td></tr></table>	Value	Meaning	0x0000	Monochrome/printer adapter	0x0001	Color graphics adapter	0x0002	Enhanced graphics adapter
Value	Meaning								
0x0000	Monochrome/printer adapter								
0x0001	Color graphics adapter								
0x0002	Enhanced graphics adapter								

	0x0003	Video graphics array or IBM Personal System/2 display adapter
display	Specifies the display/monitor type. It has one of the following values:	
	Value	Meaning
	0x0000	Monochrome display
	0x0001	Color display
	0x0002	Enhanced color display
	0x0003	8503 monochrome display
	0x0004	8512, 8513, or 8514 color display
cbMemory	Specifies the amount of memory on the adapter in bytes.	

Comments

MS OS/2 derives the **adapter** and **display** values for the video display configuration by using various tests, including checking the switch settings on the card.

Example

This example calls **VioGetConfig** to determine if the display type is an enhanced color display:

```

VIOCONFIGINFO vioinConfig;
vioinConfig.cb = sizeof(vioinConfig);      /* structure length */
VioGetConfig(0,                             /* Reserved */
             &vioinConfig,                 /* configuration data */
             0);                            /* video handle */
if (vioinConfig.display == 2)
    VioWrtTTY("Enhanced color display\n\r", 24, 0);

```

See Also

VioGetMode, **VioGetState**

- **USHORT VioGetCp**(*usReserved*, *pIdCodePage*, *hvio*)
- USHORT** *usReserved*; Must be zero
- PUSHORT** *pIdCodePage*; code-page identifier
- HVIO** *hvio*; video handle

The **VioGetCp** function retrieves the code-page identifier of the current video code page. The video code page defines the character set being used

to display text on the screen. If the identifier is 0x0000, the display adapter's built-in code page is being used. Any other value identifies a code page that has been set by using the **VioSetCp** function or inherited from the parent process.

Parameter	Description
<i>usReserved</i>	Specifies a reserved value. It must be zero.
<i>pIdCodePage</i>	Points to a variable that receives the code-page identifier.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

Example

This example calls **VioGetCp** to obtain the current system code page:

```
USHORT idCodePage;  
VioGetCp(0,                                /* Must be zero */  
        &idCodePage,                       /* code-page identifier */  
        0);                                /* video handle */
```

See Also

DosGetCp, DosSetCp, VioSetCp

■ **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.

Parameter	Description
<i>pusRow</i>	Points to a variable that receives the current row position of the cursor.
<i>pusColumn</i>	Points to a variable that receives the current column position of the cursor.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

Example

This example calls **VioGetCurPos** to retrieve the current row-and-column position of the cursor:

```
USHORT usRow, usCol;
VioGetCurPos(&usRow,          /* address of row */
              &usCol,          /* address of column */
              0);              /* video handle */
```

See Also

VioGetCurType, **VioSetCurPos**

■ **USHORT VioGetCurType**(*pviociCursor*, *hvio*)
PVIOCURSORINFO *pviociCursor*; pointer to buffer for cursor type
HVIO *hvio*; video handle

The **VioGetCurType** function retrieves information about the cursor type. The cursor type 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.

Parameter	Description
<i>pviociCursor</i>	Points to a VIOCURSORINFO structure that receives information about the cursor type. For a full description, see the following “Structures” section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

Structures

The **VIOCursorInfo** structure pointed to by the *pvioCursor* parameter has the following format:

```
typedef struct _VIOCursorInfo {
    USHORT yStart;
    USHORT cEnd;
    USHORT cx;
    USHORT attr;
} VIOCursorInfo;
```

Field	Description
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 pixels (for graphics mode).
attr	Specifies the attribute of the cursor. If this field is 0xFFFF, the cursor is hidden (not visible on the screen). Any other value specifies the current character attribute of the cursor.

Example

This example calls **VioGetCurType** to retrieve the current cursor type, changes the attribute to hidden or non-hidden (the opposite of what it was), and calls **VioSetCurType** to set the new cursor type:

```
VIOCursorInfo vioCursor;
VioGetCurType(&vioCursor, 0); /* Gets current cursor type */
vioCursor.attr = /* Flips attribute to hidden/nonhidden */
    (vioCursor.attr == -1) ? 0 : -1;
VioSetCurType(&vioCursor, 0); /* Sets the new cursor type */
```

See Also

VioGetCurPos, VioSetCurType

- **USHORT VioGetFont**(*pviofiFont*, *hvio*)
PVIOFONTINFO *pviofiFont*; pointer to buffer for font
HVIO *hvio*; video handle

The **VioGetFont** function retrieves a specified font. A font consists of one or more bitmaps that define the shape of characters when displayed on the screen. The **VioGetFont** function retrieves a copy of either the current font or a font from the video display adapter's ROM.

Parameter	Description
<i>pviofiFont</i>	Points to a VIOFONTINFO structure that specifies the request type and receives the font information. For a full description, see the following "Structures" section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_COL**
The column value is invalid.
- ERROR_VIO_FONT**
No font is available to support the mode.
- ERROR_VIO_INVALID_PARMS**
The reserved word is not zero.
- ERROR_VIO_ROW**
The row value is invalid.

Structures

The **VIOFONTINFO** structure pointed to by the *pviofiFont* parameter has the following format:

```
typedef struct _VIOFONTINFO {
    USHORT cb;
    USHORT type;
    USHORT cxCell;
    USHORT cyCell;
    ULONG pbData;
    USHORT cbData;
} VIOFONTINFO;
```

Field	Description
cb	Specifies the length of the structure in bytes. The length must be set to 0x000E bytes.
type	Specifies the request type. This field must be set to VGFL_GETCURFONT to retrieve the current font. It must be set to VGFL_GETROMFONT to retrieve a ROM font.
cxCell	Specifies the width in pixels of each character cell in the font.
cyCell	Specifies the height in pixels of each the character cell in the font.
pbData	Points to the buffer that receives the requested font table, or can be set to 0x00000000 to direct VioGetFont to supply an address. In the latter case, the function copies the address of the font to the pbData field. The address specifies either a RAM or ROM address, depending on the request type.
cbData	Specifies the length in bytes of the font.

When requesting a ROM font, the **cxCell** and **cyCell** fields must be set before calling the function. These fields identify the font to be retrieved.

Comments

Although the **VioGetFont** function can retrieve fonts for many display adapters, the fonts for some adapters are not available. In most case, 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 area. The current font is defined by the most recent **DosSetCp** or **VioSetCp** function, or 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 of a character cell in points. The **pbData** field points to the actual font:

```

VIOFONTINFO viofiFont;
viofiFont.cb = sizeof(viofiFont); /* length of structure */
viofiFont.type = VGFI_GETCURFONT; /* Gets the current font */
viofiFont.cxCell = 0; /* Clears the columns */
viofiFont.cyCell = 0; /* Clears the rows */
viofiFont.pbData = 0L; /* address of data area */
viofiFont.cbData = 0L; /* length of data area */
VioGetFont(&viofiFont, 0);

```

See Also

VioSetCp, VioSetFont

■ **USHORT VioGetMode**(*pviomiMode*, *hvio*)
PVIOMODEINFO *pviomiMode*; pointer to buffer for mode
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), the width and height of the screen in character cells, and the width and height of the screen in pixels.

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

Parameter	Description
<i>pviomiMode</i>	Points to a VIOMODEINFO structure that receives the screen mode information. For a full description, see the following “Structures” section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_INVALID_LENGTH
The control block length is invalid.

Structures

The **VIOMODEINFO** structure pointed to by the *pviomiMode* parameter has the following format:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR fbType;
    UCHAR color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR fmt_ID;
    UCHAR attrib;
} VIOMODEINFO;
```

Field	Description										
cb	Specifies the length in bytes of the data structure. This field must be set to 0x000E bytes before calling the function.										
fbType	Specifies the screen mode. It is one of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>Monochrome/printer adapter enabled.</td></tr> <tr> <td>VGMT_OTHER</td><td>Non-monochrome/printer adapter enabled.</td></tr> <tr> <td>VGMT_GRAPHICS</td><td>Graphics mode enabled.</td></tr> <tr> <td>VGMT_DISABLEBURST</td><td>Color burst disabled.</td></tr> </table>	Value	Meaning	0x0000	Monochrome/printer adapter enabled.	VGMT_OTHER	Non-monochrome/printer adapter enabled.	VGMT_GRAPHICS	Graphics mode enabled.	VGMT_DISABLEBURST	Color burst disabled.
Value	Meaning										
0x0000	Monochrome/printer adapter enabled.										
VGMT_OTHER	Non-monochrome/printer adapter enabled.										
VGMT_GRAPHICS	Graphics mode enabled.										
VGMT_DISABLEBURST	Color burst disabled.										
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: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>2 colors</td></tr> <tr> <td>0x0002</td><td>4 colors</td></tr> <tr> <td>0x0004</td><td>16 colors</td></tr> </table>	Value	Meaning	0x0001	2 colors	0x0002	4 colors	0x0004	16 colors		
Value	Meaning										
0x0001	2 colors										
0x0002	4 colors										
0x0004	16 colors										
col	Specifies the number of text columns.										
row	Specifies the number of text rows.										
hres	Specifies the number of pixel columns (horizontal resolution).										
vres	Specifies the number of pixel rows (vertical resolution).										
fmt_ID	Specifies a reserved value. It is be zero.										
attrib	Specifies a reserved value. It is be zero.										

Example

This example calls **VioGetMode** to retrieve the mode data for the screen:

```
VIOMODEINFO viomiMode;
viomiMode.cb = sizeof(viomiMode);
VioGetMode(&viomiMode, 0);
if (vmiMode.fbType == 0) {
    VioWrtTty("Monochrome display\n\r", 21, 0);
}
```

See Also

VioGetState, **VioSetMode**

- **USHORT VioGetPhysBuf(*pviopbBuffer*, *usReserved*)**
PVIOPHYSBUF *pviopbBuffer*; pointer to buffer 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 actual content of the screen depends on the current screen mode and the display adapter type.

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

Parameter	Description
<i>pviopbBuffer</i>	Points to a VIOPHYSBUF structure that specifies the physical video-buffer address and length, and receives the selector(s) used to address the video buffer. For a full description, see the following “Structures” section.
<i>usReserved</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_IN_BG**
This function is only permitted when this process is in the foreground.
- ERROR_VIO_INVALID_HANDLE**
The video device handle is invalid.

Structures

The **VIOPHYSBUF** structure pointed to by the *pviopbBuffer* parameter has the following format:

```
typedef struct _VIOPHYSBUF {  
    PBYTE pBuf;  
    ULONG cb;  
    SEL   asel[1];  
} VIOPHYSBUF;
```

Field	Description
pBuf	Specifies the 32-bit physical address of the first byte in the physical video buffer. The address must be from 0x000A0000 to 0x000BFFFF. The address to be used depends on the display adapter and the video mode. For more information, see Appendix B, “Devices.”
cb	Specifies the length in bytes of the physical video buffer.
asel[1]	Specifies an 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.

The actual size of the **asel** field depends on the size of physical memory. The program must ensure that there is adequate space to receive all selectors.

Comments

Since the physical video buffer is subject to change by the current foreground process, it is recommended that the buffer be accessed only by the foreground process. The **VioScrLock** function can be used to ensure that the process has complete control of the physical buffer.

Example

This example locks the screen, calls **VioGetPhysBuf** to retrieve the address of the physical screen buffer, unlocks the screen, and assigns the address of the physical screen buffer to a pointer:


```

VIOPHYSBUF viopbBuf;
PCH pchScreen;
USHORT fStatus;
viopbBuf.pBuf = 0xB000L;
viopbBuf.cb = 4000;
VioScrLock(LOCKIO_WAIT, &fStatus, 0);
VioGetPhysBuf(&viopbBuf, 0);
VioScrUnLock(0);
pchScreen = MAKEPCH(viopbBuf.asel[0], 0);

```

See Also

VioGetBuf, VioScrLock, VioScrUnLock, VioShowBuf

■ **USHORT VioGetState(*pvoidState*, *hvio*)**

PVOID *pvoidState*;

pointer to buffer for state

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.

Parameter	Description
<i>pvoidState</i>	Points to a data structure that receives the state information. The structure type depends on the request type specified. For a full description of the structures, see the following “Structures” section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_INVALID_LENGTH

The parameter length is invalid.

Structures

The *pvoidState* parameter points to one of three data structures: **VIOPALSTATE**, **VIOOVERSCAN**, or **VIOINTENSITY**. The format and the content of the structure depends on the request type that is specified by the **type** field in each structure.

The **VIOPALSTATE** structure has the following format:

```
typedef struct _VIOPALSTATE {
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;
```

Field	Description
cb	Specifies the length of the structure in bytes. The length determines how many palette registers are retrieved. The maximum length is 38 bytes (0x0026) 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.

The **VIOOVERSCAN** structure has the following format:

```
typedef struct _VIOOVERSCAN {
    USHORT cb;
    USHORT type;
    USHORT color;
} VIOOVERSCAN;
```

Field	Description
cb	Specifies the length of the structure in bytes.
type	Specifies the request type. To retrieve the overscan (border) color this field must be set to 0x0001.
color	Specifies the color value.

The **VIOINTENSITY** structure has the following format:

```
typedef struct _VIOINTENSITY {
    USHORT  cb;
    USHORT  type;
    USHORT  fs;
} VIOINTENSITY;
```

Field	Description
cb	Specifies the length of the structure in bytes.
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.

In each case, the **cb** and **type** fields must be set before calling the function.

Comments

Not all **type** field values are valid for all screen modes.

Example

This example calls the **VioGetState** function to retrieve the settings for each of the sixteen palette registers:

```
BYTE abState[38];
PVIOPALSTATE pviopalState;
pviopalState = (PVIOPALSTATE) abState;
pviopalState->cb = sizeof(abState);           /* structure size */
pviopalState->type = 0;                        /* retrieve palette registers */
pviopalState->iFirst = 0; /* first palette register to return */
VioGetState(pviopalState, 0);
```

See Also

VioGetMode, **VioSetState**

■ **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 informed of a change of video mode. A process requests to be informed 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 allows the thread to continue (or terminates the thread, if desired).

MS OS/2 permits only one process in a screen group to make to a video-mode change request. 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. The **VioModeUndo** function can be used to relinquish ownership of the request. If the *fRelinquish* parameter is set to 0x0001, the function cancels the request and relinquishes ownership. If *fRelinquish* is 0x0000, the process retains ownership and can make the request again without competing with other processes.

Parameter	Description
<i>fRelinquish</i>	Specifies whether to retain or relinquish ownership of the request. If this parameter is 0x0000, the function retains ownership. If it is 0x0001, the function relinquishes ownership.
<i>fTerminate</i>	Specifies whether or not to terminate the thread waiting for the mode change. If this parameter is 0x0000, the thread continues and receives an error value from the VioModeWait function. If it is 0x0001, the thread terminates.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_FUNCTION_OWNED

Another process in this screen group has already called the **VioModeWait** function.

ERROR_VIO_INVALID_PARMS

Invalid parameters were specified.

ERROR_VIO_NO_MODE_THREAD

No thread has called the **VioModeWait** function.

Comments

Only the process that owns the change-mode request may called the **VioModeUndo** function.

See Also**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 directs a thread to wait for a change in the current video mode. When a change occurs, the system sets the variable pointed to by the *pfNotify* parameter to a value indicating the type of change and restores execution to the calling thread. 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 typically used by graphics programs (or text programs that access video registers directly) to restore the screen after a pop-up screen has been displayed. Pop-up screens often change the screen 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, and then is granted execution so that it may 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.

Parameter	Description
<i>fEvent</i>	Specifies the event to wait for. If this parameter is VMWR_POPUP, the function waits for a pop-up screen to close. No other request values are permitted.
<i>pfNotify</i>	Points to a variable that receives a flag that specifies the action to carry out in response to the given event. If this parameter is VMWN_POPUP, the process should restore the video mode. No other values are returned.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_FUNCTION_OWNED

Another thread has called the **VioModeWait** function.

ERROR_VIO_INVALID_PARMS

Invalid parameters are specified.

ERROR_VIO_RETURN

Another thread has called the **VioModeUndo** function.

Comments

A program should use the **VioModeWait** function if it changes the video registers directly. MS OS/2 will automatically save and restore the physical screen 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 (directly or indirectly through other functions) that may generate pop-up screens or error pop-ups. Doing so may cause a system lockout (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). The **VioModeUndo** function can be used to terminate the thread when it is no longer needed.

Programs that need to save and restore the video mode and screen before and after a screen switch should use the **VioSaveRedrawWait** function.

See Also

VioModeUndo, **VioPopUp**, **VioSaveRedrawWait**

■ **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 to the user without altering the content of the process's regular screen. Pop-up screens are typically used by background processes to display messages when the screen is not available.

The **VioPopUp** function may change the screen mode if it is not already in 25-line by 80-column text mode. Also, the pop-up screen can be opaque or transparent, as specified by the *pfWait* parameter. If the pop-up screen is opaque, the function clears the screen and moves the cursor to the upper-left corner. If the pop-up screen is transparent, the function leaves the screen and the cursor unchanged.

Once the pop-up screen is open, the process may use any of the following video 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 the system disables the keys that normally switch the system from one screen group to another. While a pop-up screen is open, the process must not attempt to access or modify the physical screen 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 a pop-up screen when one is already open, the function can wait until the previous screen is closed before opening the pop-up screen. The *pfWait* parameter is used to specify whether the function should wait or return an error value.

Parameter	Description
<i>pfWait</i>	Points to a variable that specifies whether the pop-up screen is to be opaque or transparent. It also specifies whether the function should wait for any open pop-up screen to close. It can be any combination of the following values:
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.
VP_OPAQUE	Change the screen mode to 25-by-80 text mode, clear the screen, and move the cursor to the upper-left corner of the screen.
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 the appropriate text mode already.

hvio Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_EXISTING_POPUP

A pop-up screen already exists.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_NO_POPUP

Comments

While a pop-up screen is open, the system blocks any video functions called by the process that owns the previous screen. This means any output generated by the interrupted process will not be lost. The system displays the output after the pop-up screen is closed.

Before opening a pop-up screen, the system saves the physical video buffer of the existing screen. A pop-up screen can be closed by using the **VioEndPopUp** function. When the pop-up screen is closed, **VioEndPopUp** restores the screen mode and the screen buffer. The function also restores keyboard input to the previous process and enables the screen group switching keys. In some cases, the **VioEndPopUp** function may not completely restore the screen. In such cases, the **VioModeWait** function can be used to restore the screen.

Transparent pop-up screens cannot be used if the foreground process has called the **VioSavRedrawWait** function.

If a process registers a replacement **VioPopUp** function (by using the **VioRegister** function), the system uses the replacement function only if the foreground process requests a pop-up screen. If a background process requests a pop-up screen, the system 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 it is not available:


```
USHORT fWait = VP_WAIT;
VioPopUp(&fWait, 0);
.
.          /* Message and user interaction would go here */
.
VioEndPopUp(0); /* Ends the pop-up screen          */
```

See Also

DosExecPgm, VioEndPopUp, VioGetPhysBuf

■ **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 only. It is called whenever the PRINTSCREEN key is pressed.

Parameter	Description
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_SMG_ONLY
Only the session manager may call this function.

Comments

Although the **VioPrtSc** function is reserved for system use, a process can replace it with a custom print-screen function by using the **VioRegister** function. If the **VioPrtSc** function is replaced, all processes in that screen group use the replacement function. This gives a process the capability of capturing input from the PRINTSCREEN key.

See Also

VioPrtScToggle, VioRegister

- **USHORT VioPrtScToggle(*hvio*)**
HVIO *hvio*; video handle

The **VioPrtScToggle** function enables or disables the printer echo feature.

This function is reserved for system use. The system calls the function whenever the CONTROL+PRINTSCREEN key is pressed. The first press enables the printer echo feature, the second disables it.

Parameter	Description
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE

ERROR_VIO_SMG_ONLY

Only the session manager may call this function.

Comments

Although the **VioPrtScToggle** function is reserved for system use, a process can replace it with a custom function by using the **VioRegister** function. If the **VioPrtScToggle** function is replaced, all processes in that screen group use the replacement function. This gives a process the capability of capturing input from the CONTROL+PRINTSCREEN key.

See Also

VioPrtSc

- **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.

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

Parameter	Description
<i>pchCellString</i>	Points to a buffer that receives the cell string.
<i>pcb</i>	Points to a variable that contains the length of the buffer in bytes. The length should be an even number. On return, this function copies the actual length of the string to the variable.
<i>usRow</i>	Specifies the starting row of the cell string to read.
<i>usColumn</i>	Specifies the starting column of the cell string to read.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL
The column value is invalid.

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_ROW
The row value is invalid.

Comments

If the length of the string is longer than the current line, the **VioReadCellStr** function continues reading the string at the beginning of the next line, but does not read past the end of the screen.

Example

This example calls **VioReadCellStr** to read line 0. It then writes 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

- **USHORT VioReadCharStr**(*pchString*, *pcb*, *usRow*, *usColumn*, *hvio*)
PCH *pchString*; pointer to buffer for string
PUSHORT *pcb*; pointer to variable for length
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.

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

Parameter	Description
<i>pchString</i>	Points to the buffer that receives the character string.
<i>pcb</i>	Points to a variable that contains the length of the buffer in bytes. On return, the function copies the actual length of the string to the variable.
<i>usRow</i>	Specifies the starting row of the character to be read.
<i>usColumn</i>	Specifies the starting column of the character to be read.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_COL**
The column value is invalid.
- ERROR_VIO_INVALID_HANDLE**
The video device handle is invalid.
- ERROR_VIO_ROW**
The row value is invalid.

Comments

If the character-string length is longer than the current line, the **VioReadCharStr** function continues reading the string at the beginning of the next line, but does not read past the end of the screen.

Example

This example calls the **VioReadCharStr** function to read a character string that is 80 characters long starting at row 1, column 0 of the screen. It then writes 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**

■ **USHORT VioRegister**(*pszModuleName*, *pszEntryName*, *fFunction1*, *fFunction2*)

PSZ <i>pszModuleName</i> ;	pointer to module name
PSZ <i>pszEntryName</i> ;	pointer to entry-point name
ULONG <i>fFunction1</i> ;	function flag 1
ULONG <i>fFunction2</i> ;	function flag 2

The **VioRegister** function registers an alternate video subsystem within a screen group. The **VioRegister** function temporarily replaces one or more default video functions, as specified by the *fFunction1* and *fFunction2* parameters, with the functions named by the *pszModuleName* parameter. Once **VioRegister** replaces a function, the system passes any subsequent call to the replaced function to a function in the given module. If a function is not replaced, the system continues to call the default video function.

The system passes a video function to the given module by preparing the stack and calling the function named by the *pszEntryName* parameter. The entry-point function name must be exported by the specified module. 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 function may then access any additional parameters placed on the stack by the original call.

Parameter	Description
<i>pszModuleName</i>	Points to a null-terminated string that specifies the name of the dynamic-link module that contains the replacement video functions. The string must be a valid MS OS/2 filename.
<i>pszEntryName</i>	Points to a null-terminated string that specifies the dynamic-link entry-point name of the function that replaces the specified video functions. For a full description, see the following “Comments” section.
<i>flFunction1</i>	Specifies the video function(s) to be replaced. It can be any combination of the following values:

Value	Meaning
0x00000001	Replace VioGetCurPos .
0x00000002	Replace VioGetCurType .
0x00000004	Replace VioGetMode .
0x00000008	Replace VioGetBuf .
0x00000010	Replace VioGetPhysBuf .
0x00000020	Replace VioSetCurPos .
0x00000040	Replace VioSetCurType .
0x00000080	Replace VioSetMode .
0x00000100	Replace VioShowBuf .
0x00000200	Replace VioReadCharStr .
0x00000400	Replace VioReadCellStr .
0x00000800	Replace VioWrtNChar .
0x00001000	Replace VioWrtNAttr .
0x00002000	Replace VioWrtNCell .
0x00004000	Replace VioWrtTTy .
0x00008000	Replace VioWrtCharStr .
0x00010000	Replace VioWrtCharStrAtt .
0x00020000	Replace VioWrtCellStr .
0x00040000	Replace VioScrollUp .
0x00080000	Replace VioScrollDn .
0x00100000	Replace VioScrollLf .

0x00200000	Replace VioScrollRt.
0x00400000	Replace VioSetAnsi.
0x00800000	Replace VioGetAnsi.
0x01000000	Replace VioPrtSc.
0x02000000	Replace VioScrLock.
0x04000000	Replace VioScrUnLock.
0x08000000	Replace VioSavRedrawWait.
0x10000000	Replace VioSavRedrawUndo.
0x20000000	Replace VioPopUp.
0x40000000	Replace VioEndPopUp.
0x80000000	Replace VioPrtScToggle.

fFunction2

Specifies the video function(s) to be replaced. It can be any combination of the following values:

Value	Meaning
0x00000001	Replace VioModeWait.
0x00000002	Replace VioModeUndo.
0x00000004	Replace VioGetFont.
0x00000008	Replace VioGetConfig.
0x00000010	Replace VioSetCp.
0x00000020	Replace VioGetCp.
0x00000040	Replace VioSetFont.
0x00000080	Replace VioGetState.
0x00000100	Replace VioSetState.

Return Value

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

ERROR_VIO_INVALID_ASCII

The null-terminated character-string length is invalid.

ERROR_VIO_INVALID_MASK

The replacement mask is invalid.

ERROR_VIO_REGISTER

VioRegister

Comments

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 video functions. The process can restore the default video functions by calling the **VioDeRegister** function. A process can replace video functions any number of times, but only by first restoring the default functions and then reregistering the new functions.

The entry-point function must have the following form:

```
VOID FAR FuncName(selDataSeg, usReserved1, Function, 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
-----------	-------------

<i>selDataSeg</i>	Specifies the data segment selector of the process calling the video function.
-------------------	--

<i>usReserved1</i>	Specifies a reserved value that must not be changed. It represents a return address for the system function that routes video function calls.
--------------------	---

<i>fFunction</i>	Specifies the function code of the function request. It can be any one of the following values:
------------------	---

Index	Function	Index	Function
0x0000	VioGetPhysBuf	0x0015	VioScrollRt
0x0001	VioGetBuf	0x0016	VioSetAnsi
0x0002	VioShowBuf	0x0017	VioGetAnsi
0x0003	VioGetCurPos	0x0018	VioPrtSc
0x0004	VioGetCurType	0x0019	VioScrLock
0x0005	VioGetMode	0x001A	VioScrUnLock
0x0006	VioSetCurPos	0x001B	VioSavRedrawWait
0x0007	VioSetCurType	0x001C	VioSavRedrawUndo
0x0008	VioSetMode	0x001D	VioPopUp
0x0009	VioReadCharStr	0x001E	VioEndPopUp
0x000A	VioReadCellStr	0x001F	VioPrtScToggle
0x000B	VioWrtNChar	0x0020	VioModeWait
0x000C	VioWrtNAttr	0x0021	VioModeUndo
0x000D	VioWrtNCell	0x0022	VioGetFont
0x000E	VioWrtCharStr	0x0023	VioGetConfig
0x000F	VioWrtCharStrAtt	0x0024	VioSetCp

0x0010	VioWrtCellStr	0x0025	VioGetCp
0x0011	VioWrtTTY	0x0026	VioSetFont
0x0012	VioScrollUp	0x0027	VioGetState
0x0013	VioScrollDn	0x0028	VioSetState
0x0014	VioScrollLf		

ulReserved2 Specifies a reserved value that must not be changed. It represents the return address to the program calling the specified video function.

usParam1–usParam6 Specifies up to six values passed with the original video function call. Not all requests include all six parameters since not all video functions use six parameters. The number and type of parameters used depend on each function, as specified in this manual.

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 may call a function within the same module to carry out the task. In any case, the 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 function.

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 4, “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**

■ **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 informed when the system switches screens. A process requests to be informed by calling the **VioSavRedrawWait** function. The request forces the calling thread to wait until a screen switch occurs. The **VioSavRedrawUndo** cancels the request and allows the thread to continue (or terminates the thread, if desired).

MS OS/2 permits only one process in a screen group to make a screen-switch request. 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. The **VioSavRedrawUndo** function can be used to relinquish ownership of the request. If the *fRelinquish* parameter is set to **UNDOL_GETOWNER**, the function cancels the request and relinquishes ownership. If the parameter is **UNDOL_RELEASEOWNER**, the process retains ownership and can make the request again without competing with other processes.

Parameter	Description
<i>fRelinquish</i>	Specifies whether to retain or relinquish ownership of the request. If it is UNDOL_GETOWNER , the function retains ownership. If it is UNDOL_RELEASEOWNER , the function relinquishes ownership.
<i>fTerminate</i>	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 it is UNDOK_TERMINATE , the thread terminates.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_FUNCTION_OWNED

Another process in this screen group has already called a **VioSavRedrawWait**.

ERROR_VIO_INVALID_PARMS

Invalid parameters were specified.

ERROR_VIO_NO_SAVE_RESTORE_THD

No thread has called **VioSavRedrawWait**.

Comments

Only the process that owns the change mode request may call the **VioSavRedrawUndo** function.

See Also

VioModeUndo, **VioSavRedrawWait**

—

—

—

Return Value

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

ERROR_VIO_FUNCTION_OWNED

A thread has already called **VioSavRedrawWait**.

ERROR_VIO_INVALID_PARMS

Invalid parameters are specified.

ERROR_VIO_RETURN

Another thread has called **VioSavRedrawUndo**.

Comments

Graphics programs (or text-mode programs that change the video registers directly) should use the **VioSaveRedrawWait** function to guarantee that the screen image will be saved and restored when screens are switched. Although MS OS/2 attempts to save and restore each screen, the system may not completely save and restore the screen contents and modes.

When an application is notified to save its screen image, it saves its physical screen buffer, video mode, and any other information the application needs to redraw its screen completely.

The thread that calls **VioSavRedrawWait** should carry out all tasks directly related to saving and restoring the complete 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 screen 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 (directly or indirectly through other functions) that may generate pop-up screens or error pop-ups. Doing so may cause a system lockout (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). The **VioSavRedrawUndo** function can be used to terminate the thread when it is no longer needed.

In some cases, a thread may receive a request to redraw the screen before receiving a request to save the screen. The thread must determine whether a 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

■ **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 screen buffer for a process. While the buffer is locked, no other process may lock it. This function is typically used to coordinate the output of graphics programs to prevent more than one process from writing to the physical screen buffer at the same time. The function does not actually keep processes from modifying the physical screen buffer. Instead, it indicates when the screen is locked by another process and presumably not available for writing.

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 defined by the *fWait* parameter. Processes that lock the screen should unlock the screen with the **VioScrUnlock** function as soon as they have completed the output.

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

Parameter	Description
<i>fWait</i>	Specifies whether the process is to block until the screen I/O can occur. If it is LOCKIO_NOWAIT , the function returns immediately if the screen is not available. If it is LOCKIO_WAIT , the function waits for the screen to become available.
<i>pfNotLocked</i>	Points to a variable that receives a value that specifies whether the screen is locked. This parameter is LOCK_SUCCESS if the screen is locked. It is LOCK_FAIL if the screen is not locked.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_LOCK
The screen is already locked.

ERROR_VIO_WAIT_FLAG

The *fWait* setting is invalid.

Comments

If the user requests a screen switch while the screen lock is in place, the switch is held for at least 30 seconds. If the process does not unlock the screen before 30 seconds elapses, the system freezes the process and switches the screen. The frozen process remains in the background until it is switched back.

Family API 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**

- **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*; cell to write
HVIO *hvio*; video handle

The **VioScrollDn** function scrolls the current screen down.

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

Parameter	Description
<i>usTopRow</i>	Specifies the top row of the area to scroll.
<i>usLeftCol</i>	Specifies the leftmost column of the area to scroll.
<i>usBotRow</i>	Specifies the bottom row of the area to scroll.
<i>usRightCol</i>	Specifies the rightmost column of the area to scroll.
<i>cbLines</i>	Specifies the number of lines to be inserted at the top of the screen area being scrolled. If the <i>cbLines</i> parameter equals zero, no lines are scrolled.
<i>pbCell</i>	Points to a character-attribute pair, called a cell, that fills the area left blank by the scroll.
<i>hwio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL

The left column is greater than the right column.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The top row is greater than the bottom row.

Comments

If the *usTopRow* and *usLeftCol* parameters equal zero, they identify the top-left corner of the screen. If a value greater than the maximum value is specified for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbLines*, the maximum value for that parameter is used. The maximum values permitted depend upon the actual dimensions of the screen being used.

The **VioScrollDn** function can be used 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 using the character and attribute pointed to by the *pbCell* parameter.

Example

This example creates a cell containing the space character (0x20) and a white attribute (0x07 for EGA), and calls **VioScrollDn** to clear the screen by using this cell. By changing the attribute, you could change the background color of the screen while clearing it at the same time:

```

BYTE bBlank[2];
bBlank[0] = 0x20;
bBlank[1] = 0x07;
VioScrollDn(0,
    0,
    -1,
    -1,
    -1,
    bBlank,
    0);
/* space character */
/* white attribute (EGA) */
/* top row */
/* left column */
/* bottom row */
/* right column */
/* number of lines */
/* cell to write */
/* video handle */

```

See Also

VioScrollLf, **VioScrollRt**, **VioScrollUp**

■ **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*; cell to write
HVIO *hvio*; video handle

The **VioScrollLf** function scrolls the current screen to the left.

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

Parameter	Description
<i>usTopRow</i>	Specifies the top row of the area to scroll.
<i>usLeftCol</i>	Specifies the leftmost column of the area to scroll.
<i>usBotRow</i>	Specifies the bottom row of the area to scroll.
<i>usRightCol</i>	Specifies the rightmost column of the area to scroll.
<i>cbColumns</i>	Specifies the number of blank columns to be inserted at the right. If the <i>cbColumns</i> parameter equals zero, no columns are inserted.

pbCell Points to a character-attribute pair, called a cell, that fills the area left blank by the scroll.

hwio Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL
The left column is greater than the right column.

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_ROW
The top row is greater than the bottom row.

Comments

If the *usTopRow* and *usLeftCol* parameters equal zero, they identify the top-left corner of the screen. If a value greater than the maximum value is specified for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbColumns*, the maximum value for that parameter is used. The maximum values permitted depend upon the actual dimensions of the screen being used.

The **VioScrollLf** function can be used 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 using the character and attribute 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:

```

BYTE bHeart[2];
bHeart[0] = 0x03;
bHeart[1] = 0x04;
VioScrollLf(0,
    0,
    -1,
    -1,
    10,
    bHeart,
    0);
/* heart character */
/* red attribute (EGA) */
/* top row */
/* left column */
/* bottom row */
/* right column */
/* columns */
/* cell to write */
/* video handle */

```

See Also

VioScrollDn, VioScrollRt, VioScrollUp

■ **USHORT VioScrollRt**(*usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*,
cbColumns, *pbCell*, *hvio*)

USHORT <i>usTopRow</i> ;	top row
USHORT <i>usLeftCol</i> ;	left column
USHORT <i>usBotRow</i> ;	bottom row
USHORT <i>usRightCol</i> ;	right column
USHORT <i>cbColumns</i> ;	number of blank lines
PBYTE <i>pbCell</i> ;	cell to write
HVIO <i>hvio</i> ;	video handle

The **VioScrollRt** function scrolls the current screen to the right.

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

Parameter	Description
<i>usTopRow</i>	Specifies the top row of the area to scroll.
<i>usLeftCol</i>	Specifies the leftmost column of the area to scroll.
<i>usBotRow</i>	Specifies the bottom row of the area to scroll.
<i>usRightCol</i>	Specifies the rightmost column of the area to scroll.
<i>cbColumns</i>	Specifies the number of blank columns to be inserted at the left. If the <i>cbColumns</i> parameter is zero, no columns are inserted.
<i>pbCell</i>	Points to a character-attribute pair, called a cell, that fills the area left blank by the scroll.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL
The left column is greater than the right column.

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_ROW
The top row is greater than the bottom row.

Comments

If the *usTopRow* and *usLeftCol* parameters equal zero, they identify the top-left corner of the screen. If a value greater than the maximum value is specified for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbColumns*, the maximum value for that parameter is used. The maximum values permitted depend upon the actual dimensions of the screen being used.

The **VioScrollUp** function can be used 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 using the character and attribute 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:

```

BYTE bHeart[2];
bHeart[0] = 0x03;
bHeart[1] = 0x04;
VioScrollRt(0,
    0,
    -1,
    -1,
    10,
    bHeart,
    0);
/* heart character */
/* red attribute (EGA) */
/* top row */
/* left column */
/* bottom row */
/* right column */
/* columns */
/* cell to write */
/* video handle */

```

See Also

VioScrollDn, **VioScrollLf**, **VioScrollUp**

- **USHORT VioScrollUp**(*usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, *cbLines*, *pbCell*, *hvio*)

USHORT <i>usTopRow</i> ;	top row
USHORT <i>usLeftCol</i> ;	left column
USHORT <i>usBotRow</i> ;	bottom row
USHORT <i>usRightCol</i> ;	right column
USHORT <i>cbLines</i> ;	number of blank lines
PBYTE <i>pbCell</i> ;	cell to write
HVIO <i>hvio</i> ;	video handle

The **VioScrollUp** function scrolls the current screen up.

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

Parameter	Description
<i>usTopRow</i>	Specifies the top row of the area to scroll.
<i>usLeftCol</i>	Specifies the leftmost column of the area to scroll.
<i>usBotRow</i>	Specifies the bottom row of the area to scroll.
<i>usRightCol</i>	Specifies the rightmost column of the area to scroll.
<i>cbLines</i>	Specifies the number of blank lines to be inserted at the bottom of the screen area being scrolled. If the <i>cbLines</i> parameter is zero, no lines are inserted.
<i>pbCell</i>	Points to a character-attribute pair, called a cell, that fills the area left blank by the scroll.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL

The left column is greater than the right column.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The top row is greater than the bottom row.

Comments

If the *usTopRow* and *usLeftCol* parameters equal zero, they identify the top-left corner of the screen. If a value greater than the maximum value is specified for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbLines*, the maximum value for that parameter is used. The maximum values permitted depend upon the actual dimensions of the screen being used.

The **VioScrollUp** function can be used 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 using the character and attribute pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollUp** to scroll the entire screen up and fill the area left blank by the scroll with blank spaces on a green background (on an EGA color monitor):

```
BYTE bBlank[2];
bBlank[0] = 0x20;
bBlank[1] = 0x22;
VioScrollUp(0,
            0,
            -1,
            -1,
            -1,
            bBlank,
            0);
VioSetCurPos(0, 0, 0);

/* top row      */
/* left column  */
/* bottom row   */
/* right column */
/* number of lines */
/* cell to write */
/* video handle  */
```

See Also

VioScrollDn, VioScrollLf, VioScrollRt

■ **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.

Parameter	Description
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_INVALID_HANDLE**
The video device handle is invalid.
- ERROR_VIO_UNLOCK**
The screen was not previously locked.

Example

This example locks the screen, and then calls **VioScrUnLock** to unlock the screen:

```
USHORT fNotLocked;  
VioScrLock(LOCKIO_WAIT, &fNotLocked, 0);  
.  
.  
VioScrUnLock(0);
```

See Also

VioScrLock

- **USHORT VioSetAnsi(*fAnsi*, *hvio*)**
USHORT *fAnsi*; ANSI flag
HVIO *hvio*; video handle

The **VioSetAnsi** function enables or disables processing of ANSI escape sequences. It sets or clears the ANSI flag which specifies whether **VioWrtTTY** function processes ANSI escape sequences.

Parameter	Description
<i>pfAnsi</i>	Specifies whether ANSI processing is enabled or disabled. If this parameter is ANSI_OFF, the ANSI state is disabled. If it is ANSI_ON, the ANSI state is enabled.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

Comments

When a screen group is started, ANSI support is enabled for that screen group.

Example

This example displays two identical strings. Before the first string is displayed, ANSI is turned off. As a result, the ANSI escape sequences are displayed as characters. Before the second string is displayed, ANSI is turned on, and the string is displayed in inverse (black characters on a white background):

```
VioSetAnsi(ANSI_OFF, 0); /* Sets ANSI off */
VioWrtTTY("\33[7mHello World\33[Om\n\r", 21, 0);
VioSetAnsi(ANSI_ON, 0); /* Sets ANSI on */
VioWrtTTY("\33[7mHello World\33[Om\n\r", 21, 0);
```

See Also

VioGetAnsi

- **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.

Parameter	Description
<i>usReserved</i>	Specifies a reserved value. It must be zero.
<i>idCodePage</i>	Specifies the code-page identifier. This parameter can be any code-page identifier specified in the codepage statement in the <i>config.sys</i> file. If this parameter is 0x0000, the function uses the system default code page.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_BAD_CP**
The specified code page is invalid.
- ERROR_VIO_INVALID_HANDLE**
The video device handle is invalid.

Example

This example calls **VioSetCp** to set the current-system code page to the standard U.S. code page:

```
if (VioSetCp(0, /* Must be zero */
             437, /* code-page identifier */
             0)) { /* video handle */
    VioWrtTTY("Codepage not specified in CONFIG.SYS\n\r", 38, 0);
}
```

See Also

DosSetCp, VioGetCp

■ **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.

Parameter	Description
<i>usRow</i>	Specifies the row position of the cursor, where zero is the top row.
<i>usColumn</i>	Specifies the column position of the cursor, where zero is the far-left column.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL
The column value is invalid.

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_ROW
The row value is invalid.

Example

This example calls **VioSetCurPos** to position 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

■ **USHORT VioSetCurType**(*pviociCursor*, *hvio*)
PVIOCursorInfo *pviociCursor*; pointer to buffer with cursor type
HVIO *hvio*; video handle

The **VioSetCurType** function sets the cursor type.

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

Parameter	Description
<i>pviociCursor</i>	Points to a VIOCursorInfo structure that specifies the characteristics of the cursor. For a full description, see the following "Structures" section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

ERROR_VIO_WIDTH
The cursor width is invalid.

Structures

The **VIOCursorInfo** structure pointed to by the *pviociCursor* parameter has the following format:

```

typedef struct _VIOCursorInfo {
    USHORT yStart;
    USHORT cEnd;
    USHORT cx;
    USHORT attr;
} VIOCursorInfo;

```

Field	Description
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 pixels (for graphics mode). 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.

Comments

The cursor is a shared resource for all processes in a screen group. If one process changes it, the cursor is changed for all processes.

Example

This example calls **VioSetCurType** to set the current cursor type to a 14-scan-line block cursor:

```
VIOCURSORINFO viociCursor;
viociCursor.yStart = 0;      /* beginning scan line for cursor */
viociCursor.cEnd = 13;      /* ending scan line, zero-based */
viociCursor.cx = 0;         /* default width, one character */
viociCursor.attr = 0;       /* normal attribute */
VioSetCurType(&viociCursor, 0);
```

See Also

VioGetCurType, **VioSetCurPos**

- **USHORT VioSetFont**(*pviofiFont*, *hvio*)
PVIOFONTINFO *pviofiFont*; pointer to buffer with font
HVIO *hvio*; video handle

The **VioSetFont** function sets the font to be used to display characters on the screen. A font consists of one bitmap for each character in a character set. The bitmap defines the character shape. The font must be compatible with the current screen mode; that is, the bitmap size must match the current character-cell size.

Parameter	Description
<i>pviofiFont</i>	Points to a VIOFONTINFO structure that specifies the display font. For a full description, see the following “Structures” section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_LENGTH
An invalid length was specified.

Structures

The **VIOFONTINFO** structure pointed to by the *pviofiFont* parameter has the following format:

```
typedef struct _VIOFONTINFO {  
    USHORT cb;  
    USHORT type;  
    USHORT cxCell;  
    USHORT cyCell;  
    ULONG pbData;  
    USHORT cbData;  
} VIOFONTINFO;
```

Field	Description
cb	Specifies the length of the structure in bytes. The length must be set to 0x000E bytes.
type	Specifies the request type. It must be set to 0x0000.
cxCell	Specifies the width in pixels of each character cell in the font.
cyCell	Specifies the height in pixels of each the character cell in the font.
pbData	Points to the buffer that contains the font table. The format of the font table depends on the display adapter and screen mode. For more information, see Appendix B, “Devices.”
cbData	Specifies the length in bytes of the font.

Comments

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.

See Also

VioGetCp, **VioGetFont**

- **USHORT VioSetMode**(*pviomiMode*, *hvio*)
PVIOMODEINFO *pviomiMode*; pointer to buffer with 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), the width and height of the screen in character cells, and the width and height of the screen in pixels. **VioSetMode** also initializes the cursor position and type, but does not clear the screen.

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

Parameter	Description
<i>pviomiMode</i>	Points to the PVIOMODEINFO structure that specifies the screen mode. For a full description, see the following "Structures" section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_INVALID_HANDLE**
The video device handle is invalid.
- ERROR_VIO_INVALID_LENGTH**
The structure length is invalid.
- ERROR_VIO_MODE**
The screen mode is unsupported.

Structures

The **VIOMODEINFO** structure pointed to by the *pviomiMode* parameter has the following format:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR fbType;
    UCHAR color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR fmt_ID;
    UCHAR attrib;
} VIOMODEINFO;
```

Field	Description										
cb	Specifies the length in bytes of the structure. This field must be set to 0x000E bytes before calling the function.										
fbType	Specifies the screen mode. It is one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Monochrome/printer adapter enabled.</td></tr><tr><td>0x0001</td><td>Non-monochrome/printer adapter enabled.</td></tr><tr><td>0x0002</td><td>Graphics mode enabled.</td></tr><tr><td>0x0004</td><td>Color burst disabled.</td></tr></table>	Value	Meaning	0x0000	Monochrome/printer adapter enabled.	0x0001	Non-monochrome/printer adapter enabled.	0x0002	Graphics mode enabled.	0x0004	Color burst disabled.
Value	Meaning										
0x0000	Monochrome/printer adapter enabled.										
0x0001	Non-monochrome/printer adapter enabled.										
0x0002	Graphics mode enabled.										
0x0004	Color burst disabled.										
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: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>2 colors</td></tr><tr><td>0x0002</td><td>4 colors</td></tr><tr><td>0x0004</td><td>16 colors</td></tr></table>	Value	Meaning	0x0001	2 colors	0x0002	4 colors	0x0004	16 colors		
Value	Meaning										
0x0001	2 colors										
0x0002	4 colors										
0x0004	16 colors										
col	Specifies the number of text columns.										
row	Specifies the number of text rows.										
hres	Specifies the number of pixel columns (horizontal resolution).										

vres	Specifies the number of pixel rows (vertical resolution).
fmt_ID	Specifies a reserved value. It is be zero.
attrib	Specifies a reserved value. It is be zero.

Comments

Note that not all combinations of screen-mode values are valid for all display devices.

Example

This example calls **VioGetMode** to retrieve the current display mode, changes the mode, and calls **VioSetMode** to enable the new display mode. If the current display mode has 25 rows, it is switched to 43 rows. If the current display has 43 rows, it is switched to 25 rows:

```
VIOMODEINFO viomiMode;  
viomiMode.cb = sizeof(viomiMode);  
VioGetMode(&viomiMode, 0);  
viomiMode.row = (viomiMode.row == 43) ? 25 : 43;  
VioSetMode(&viomiMode, 0);
```

See Also

VioGetMode, **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.

Parameter	Description
<i>pvoidState</i>	Points to a structure that contains the request type and the values to be set. For a full description of the structures for each of these request types, see the following "Structures" section.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

Structures

The *pvoidState* parameter points to one of three data structures: **VIOPALSTATE**, **VIOOVERSCAN**, or **VIOINTENSITY**. The format and the content of the structure depends on the request type which is specified as the **type** field in each structure.

The **VIOPALSTATE** structure has the following format:

```
typedef struct _VIOPALSTATE {
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;
```

Field	Description
cb	Specifies the length of the structure in bytes. The length determines how many palette registers are to be set. The maximum length is 38 bytes (0x0026) for 16 registers.
type	Specifies the request type. To set the palette register state, this field must be set to 0x0000.
iFirst	Specifies the first palette register to be set. It must be a value from 0x0000 to 0x000F. The function sets the palette registers in sequential order. The number of registers set depends on the structure size specified by the cb field.
acolor[1]	Specifies the array that contains the color values for the palette registers.

The **VIOOVERSCAN** structure has the following format:

```
typedef struct _VIOOVERSCAN {
    USHORT cb;
    USHORT type;
    USHORT color;
} VIOOVERSCAN;
```

VioSetState

Field	Description
cb	Specifies the length of the structure in bytes.
type	Specifies the request type. To set the overscan (border) color, this field must be set to 0x0001.
color	Specifies the color value.

The **VIOINTENSITY** structure has the following format:

```
typedef struct _VIOINTENSITY {
    USHORT cb;
    USHORT type;
    USHORT fs;
} VIOINTENSITY;
```

Field	Description
cb	Specifies the length of the structure in bytes.
type	Specifies the request type. To set the blink/background intensity switch, it 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.

Comments

Not all request type values are valid for all screen modes.

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 pviopalState;
PVIOPALSTATE pviopalState;
USHORT usTmp;
pviopalState = (PVIOPALSTATE) abState;
pviopalState->cb = sizeof(usState);
pviopalState->type = 0;          /* Retrieves palette registers */
pviopalState->iFirst = 0;        /* first register to retrieve */
VioGetState(pviopalState, 0);   /* Retrieves current settings */
usTmp = pviopalState->acolor[0]; /* Swaps #0 and #7 */
pviopalState->acolor[0] = pviopalState->acolor[7];
pviopalState->acolor[7] = usTmp;
VioSetState(pviopalState, 0);   /* Enables the new settings */
```


See Also

VioGetState, VioSetMode

■ **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.

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

Parameter	Description
<i>offLVB</i>	Specifies the offset into the logical video buffer at which the update to the screen is to start.
<i>cbOutput</i>	Specifies the length in bytes of the area to be updated on the screen.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

Comments

The logical video buffer may be used to directly manipulate displayed information. The **VioShowBuf** function must be called to update the physical screen with the logical video buffer.

Example

This example retrieves the address of the logical video buffer, and after making changes to that buffer, it calls the **VioShowBuf** function to update the video buffer from the logical video buffer:

```
PBYTE pbLVB;
USHORT cbLVB;
VioGetBuf((PULONG) &pbLVB, &cbLVB, 0);
.
.
VioShowBuf(0,          /* offset into logical video buffer */
            cbLVB,      /* length */
            0);         /* video handle */
```

See Also

VioGetBuf, VioGetPhysBuf

- **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 attribute it is displayed with.

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

Parameter	Description
<i>pchCellString</i>	Points to the cell string to be written.
<i>cbCellString</i>	Specifies the length in bytes of the cell string. The length should be an even number.
<i>usRow</i>	Specifies the row at which to start writing the string.
<i>usColumn</i>	Specifies the column at which to start writing the string.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL
The column value is invalid.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The row value is invalid.

Comments

If the length of the string is longer than can fit on the current line, the **VioWrtCellStr** function continues writing the string at the beginning of the next line, but does not write beyond the end of the screen.

Example

This example calls the **VioWrtCellStr** function to display the string "Hello World!", using 12 different attributes:

```
CHAR achCells[] = "H\1e\21\31\4o\5 \6W\7o\10r\111\13d\14!";
.
.
VioWrtCellStr(achCells,          /* character-attribute string */
sizeof(achCells),              /* length of string          */
10,                            /* row                       */
35,                            /* column                   */
0);                            /* video handle              */
```

See Also

VioReadCellStr, **VioWrtCharStr**, **VioWrtTTY**

- **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 attributes already in the screen to display the new characters.

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

VioWrtCharStr

Parameter	Description
<i>pchString</i>	Points to the character string to be written.
<i>cbString</i>	Specifies the length in bytes of the character string.
<i>usRow</i>	Specifies the row at which to start writing the string.
<i>usColumn</i>	Specifies the column at which to start writing the string.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL

The column value is invalid.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The row value is invalid.

Comments

If the length of the string is longer than can fit on the current line, the **VioWrtCharStr** function continues writing at the beginning of the next line, but does not write past the end of the screen.

Example

This example calls the **VioWrtCharStr** function to display the string "Hello World!" on the screen at row 12, column 30:

```
VioWrtCharStr("Hello World!",          /* string to display */
13,                                     /* length of string */
12,                                     /* row */
30,                                     /* column */
0);                                     /* video handle */
```

See Also

VioReadCharStr, **VioWrtCharStr**, **VioWrtTTY**

- **USHORT VioWrtCharStrAtt**(*pchString*, *cbString*, *usRow*, *usColumn*, *pbAttr*, *hvio*)

PCH <i>pchString</i> ;	pointer to string to write
USHORT <i>cbString</i> ;	length of string
USHORT <i>usRow</i> ;	starting position (row)
USHORT <i>usColumn</i> ;	starting position (column)
PBYTE <i>pbAttr</i> ;	attribute to be used
HVIO <i>hvio</i> ;	video handle

The **VioWrtCharStrAtt** function writes a character string to the screen using the specified attribute for each character in the string.

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

Parameter	Description
<i>pchString</i>	Points to the string to be written.
<i>cbString</i>	Specifies the length in bytes of the character string.
<i>usRow</i>	Specifies the row at which to start writing the string.
<i>usColumn</i>	Specifies the column at which to start writing the string.
<i>pbAttr</i>	Points to a variable that contains the attribute to be used for each character in the string.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

- ERROR_VIO_COL**
The column value is invalid.
- ERROR_VIO_INVALID_HANDLE**
The video device handle is invalid.
- ERROR_VIO_ROW**
The row value is invalid.

Comments

If the length of the string is longer than can fit on the current line, the **VioWrtCharStrAtt** function continues writing at the beginning of the next line, but does not write past the end of the screen.

Example

This example calls the **VioWrtCharStrAtt** function 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 */
13, /* length of string */
12, /* row */
35, /* column */
&bhAttr, /* address of attribute */
0); /* video handle */
```

See Also

VioWrtCharStr, **VioWrtNAttr**, **VioWrtTTy**

- **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 an attribute to the screen a specified number of times.

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

Parameter	Description
<i>pbAttr</i>	Points to a variable that contains the attribute to be written.
<i>cb</i>	Specifies the number of times to write the attribute.
<i>usRow</i>	Specifies the row at which to start writing the string.
<i>usColumn</i>	Specifies the column at which to start writing the string.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL

The column value is invalid.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The row value is invalid.

Comments

If the attribute is repeated more times than can fit on the current line, the **VioWrtNAttr** function continues writing the attribute at the beginning of the next line, but does not write past the end of the screen.

Example

This example calls the **VioWrtNAttr** function to change all the 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 to write */
            0,              /* row */
            0,              /* column */
            0);             /* video handle */
```

See Also

VioWrtCharStrAtt, **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 consists of two unsigned byte values that specify the character and attribute to be written.

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

Parameter	Description
<i>pbCell</i>	Points to the cell to be written.
<i>cb</i>	Specifies the number of times to write the cell.

<i>usRow</i>	Specifies the row at which to start writing the cell.
<i>usColumn</i>	Specifies the column at which to start writing the cell.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL

The column value is invalid.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The row value is invalid.

Comments

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.

Example

This example calls the **VioWrtNCell** function to fill the screen with green capital letter A's (for an EGA monitor):

```
BYTE abAttribute[2];          /* character-attribute cell */
abAttribute[0] = 'A';         /* character (letter A)    */
abAttribute[1] = 0x02;        /* attribute (green)       */
VioWrtNCell(abAttribute,      /* address of attribute    */
            80 * 25,          /* number to write         */
            0,                /* row                     */
            0,                /* column                  */
            0);               /* video handle            */
```

See Also

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 attribute already on the screen to display the new character.

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

Parameter	Description
<i>pchChar</i>	Points to the character to be written.
<i>cb</i>	Specifies the number of times to write the character.
<i>usRow</i>	Specifies the row at which to start writing the character string.
<i>usColumn</i>	Specifies the column at which to start writing the character string.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_COL

The column value is invalid.

ERROR_VIO_INVALID_HANDLE

The video device handle is invalid.

ERROR_VIO_ROW

The row value is invalid.

Comments

If the character is repeated more times than can fit on the current line, the **VioWrtNChar** function continues writing the character at the beginning of the next line, but does not write past the end of the screen.

Example

This example calls the **VioWrtNChar** function to fill the screen with capital letter A's:

```
VioWrtNChar("A",          /* address of character */
            80 * 25,       /* number to write      */
            0,             /* row                  */
            0,             /* column               */
            0);            /* video handle         */
```

See Also

VioWrtNCell

- **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 on the next line. If it reaches the end of the last line on the screen, it scrolls the screen and continues on a new line.

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

Parameter	Description
<i>pchString</i>	Points to the string to be written.
<i>cbString</i>	Specifies the length in bytes of the character string.
<i>hvio</i>	Specifies a reserved value. It must be zero.

Return Value

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

ERROR_VIO_INVALID_HANDLE
The video device handle is invalid.

Comments

For some ASCII values, **VioWrtTTY** carries out an action rather than displays a character. The following list describes the action taken when the given ASCII value is in the string:

Value	Meaning
0x08	BACKSPACE. Move the cursor left one position, without deleting any character that is under the cursor. If the cursor is at the beginning of the line, no action is taken.
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 with the **VioSetAnsi** function, **VioWrtTTY** processes any ANSI escape sequences in the string.

Example

The following example writes a message to the screen and beeps the system speaker:

```
VioWrtTTY("File not found\r\n\007", 17, 0);
```

See Also

VioSetCurPos, **VioWrtCellStr**, **VioWrtCharStr**

Chapter 4

Input-and-Output Control Functions

4.1	Introduction	393
4.2	Category and Function Codes	393
4.3	Physical Disk Control	394
4.4	Category and Function Code	394
4.5	Functions	398

4.1 Introduction

This appendix describes the input-and-output control functions, also called **IOCtl** functions. The input-and-output control functions let a program send commands to and retrieve data from a device driver by using the **DosDevIOCtl** function. This function sends the specified codes and data directly to the given device driver. The driver then carries out the specified action.

Input-and-output control functions are typically used to access information about or get data from a device driver that is not available through the standard MS OS/2 functions. For example, input-and-output control functions can be used to set the baud rate of a serial port or read input from the mouse.

4.2 Category and Function Codes

Each input-and-output control function has a category code and a function code. The category code generally defines the type of device to be accessed. MS OS/2 has several predefined categories. 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
0x0007	Mouse control
0x0008	Disk/diskette control
0x0009	Physical disk control
0x000A	Character monitor control
0x000B	General device control

In general, all category codes in the range 0x0000 to 0x007F are reserved for predefined categories. A device driver may also permit additional categories, but these must be explicitly defined by the device and be in the range 0x0080 to 0x00FF.

The function code defines the action to carry out. The function code typically specifies actions such as reading from or writing to the device and retrieving or setting the device modes. Function codes range in value from 0x0000 to 0x001F and are combined with one or more of the following values:

Value	Meaning
0x0020	The function retrieves data or information from the device. If 0x0020 is not part of the code, the function sends data or commands to the device.
0x0040	The function passes the command to the device driver. If 0x0040 is not part of the code, MS OS/2 intercepts the command.
0x0080	The function ignores the command if the device driver does not support it. If 0x0080 is not given as part of the code, the function returns an error code if the command is not supported.

The number and meaning of each function code depend on the device driver and the category.

4.3 Physical Disk Control

The physical disk control functions (category 0x0009) access physical, partitionable fixed disks. These functions require the physical disk number to identify the given disk. The physical disk number is the number the device driver uses to identify the disk. Other identifying numbers, such as the logical drive number maintained by MS OS/2, cannot be used.

The physical disk control functions retrieve data or carry out actions for the entire partitionable fixed disk. Direct track and sector input-and-output begins at the beginning of the physical drive instead of at the beginning of an extended volume. The GETPHYSDEVPARAMS function (0x0009,0x0063) describes the entire physical device, not just an extended volume.

4.4 Category and Function Code

The following list gives the input-and-output control functions in order of category and function code and shows the corresponding function name:

Category, Function	Function Name
Serial Device Control	
0x0001,0x0041	SETBAUDRATE
0x0001,0x0042	SETLINECTRL
0x0001,0x0044	TRANSMITIMM
0x0001,0x0045	SETBREAKOFF
0x0001,0x0046	SETMODEMCTRL
0x0001,0x004B	SETBREAKON
0x0001,0x0047	STOPTRANSMIT
0x0001,0x0048	STARTTRANSMIT
0x0001,0x0053	SETDCBINFO
0x0001,0x0061	GETBAUDRATE
0x0001,0x0062	GETLINECTRL
0x0001,0x0064	GETCOMMSTATUS
0x0001,0x0065	GETLINESTATUS
0x0001,0x0066	GETMODEMOUTPUT
0x0001,0x0067	GETMODEMINPUT
0x0001,0x0068	GETINQUECOUNT
0x0001,0x0069	GETOUTQUECOUNT
0x0001,0x006D	GETCOMMERROR
0x0001,0x0072	GETCOMMEVENT
0x0001,0x0073	GETDCBINFO
Screen/Pointer-Draw Control	
0x0003, 0x0072	GETPTRDRAWADDRESS
Keyboard Control	
0x0004,0x0050	SETTRANSTABLE
0x0004,0x0051	SETINPUTMODE
0x0004,0x0052	SETINTERIMFLAG
0x0004,0x0053	SETSHIFTSTATE
0x0004,0x0054	SETTYPAMATICRATE

0x0004,0x0055	SETFGNDSCREENGRP
0x0004,0x0056	SETSESMGRHOTKEY
0x0004,0x0057	SETFOCUS
0x0004,0x0071	GETINPUTMODE
0x0004,0x0072	GETINTERIMFLAG
0x0004,0x0073	GETSHIFTSTATE
0x0004,0x0074	READCHAR
0x0004,0x0075	PEEKCHAR
0x0004,0x0076	GETSESMGRHOTKEY
0x0004,0x0077	GETKEYBDTYPE

Printer Control

0x0005,0x0042	SETFRAMECTL
0x0005,0x0044	SETINFINITERETRY
0x0005,0x0046	INITPRINTER
0x0005,0x0062	GETFRAMECTL
0x0005,0x0064	GETINFINITERETRY
0x0005,0x0066	GETPRINTERSTATUS

Mouse Control

0x0007,0x0050	ALLOWPTRDRAW
0x0007,0x0051	UPDATEDISPLAYMODE
0x0007,0x0052	SCREENSWITCH
0x0007,0x0053	SETSCALEFACTORS
0x0007,0x0054	SETEVENTMASK
0x0007,0x0055	SETHOTKEYBUTTON
0x0007,0x0056	SETPTRSHAPE
0x0007,0x0057	DRAWPTR
0x0007,0x0058	REMOVEPTR
0x0007,0x0059	SETPTRPOS
0x0007,0x005A	SETPROTDRAWADDRESS
0x0007,0x005B	SETREALDRAWADDRESS

0x0007,0x005C	SETMOUSTATUS
0x0007,0x0060	GETBUTTONCOUNT
0x0007,0x0061	GETMICKEYCOUNT
0x0007,0x0062	GETMOUSTATUS
0x0007,0x0063	READEVENTQUE
0x0007,0x0064	GETQUESTATUS
0x0007,0x0065	GETEVENTMASK
0x0007,0x0066	GETSCALEFACTORS
0x0007,0x0067	GETPTRPOS
0x0007,0x0068	GETPTRSHAPE
0x0007,0x0069	GETHOTKEYBUTTON

Disk/Diskette Control

0x0008,0x0000	LOCKDRIVE
0x0008,0x0001	UNLOCKDRIVE
0x0008,0x0002	REDETERMINEMEDIA
0x0008,0x0003	SETLOGICALMAP
0x0008,0x0020	BLOCKREMOVABLE
0x0008,0x0021	GETLOGICALMAP
0x0008,0x0043	SETDEVICEPARAMS
0x0008,0x0044	WRITETRACK
0x0008,0x0045	FORMATVERIFY
0x0008,0x0063	GETDEVICEPARAMS
0x0008,0x0064	READTRACK
0x0008,0x0065	VERIFYTRACK

Physical Disk Control

0x0009,0x0000	LOCKPHYSDRIVE
0x0009,0x0001	UNLOCKPHYSDRIVE
0x0009,0x0044	WRITEPHYSTRACK
0x0009,0x0063	GETPHYSDEVICEPARAMS
0x0009,0x0064	READPHYSTRACK

Character Monitor Control

General Device Control

0x000B,0x0002	FLUSHOUTPUT
---------------	-------------

0x000B,0x0060 QUERYMONSUPPORT

This section lists the input-and-output control functions in alphabetical order. Each function's syntax is given, and the parameters and return values are described.

- The **ALLOWPTRDRAW** function notifies the mouse device driver that a screen group switch has been completed and the pointer may now be drawn.

Return Value

```

int DosDevIOCtl(pfNonRemovable, pbCommand, 0x0020, 0x0008, hDevice)
PBYTE pfNonRemovable;           pointer to removable/non-removable flag
PBYTE pbCommand;               pointer to variable with command
HFILE hDevice;                 device handle

```

398

Parameter	Description
<i>pfNonRemovable</i>	Points to a variable that receives the medium type. The variable is 0x0000 if the medium is removable; otherwise it is 0x0001.
<i>pbCommand</i>	Points to a variable that contains a reserved value. It must be zero.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

- **int DosDevIOctl(0L, 0L, 0x0057, 0x0007, *hDevice*)**
HFILE *hDevice*; device handle

The 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, then it is drawn, but if the pointer was not in the released rectangle, the pointer-draw operation takes place.

Parameter	Description
<i>hDevice</i>	Identifies the mouse device that will receive the device-control function. The handle must have been previously created using the DosOpen function.

Return Value

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

- **int DosDevIOctl(0L, *pbCommand*, 0x0001, 0x000B, *hDevice*)**
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The FLUSHINPUT function flushes the input buffer.

FLUSHINPUT

Parameter	Description
<i>pbCommand</i>	Points to a variable that contains a reserved value. It must be zero.
<i>hDevice</i>	Identifies the device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

FLUSHOUTPUT

■ **int DosDevIOCtl(0L, *pbCommand*, 0x0002, 0x000B, *hDevice*)**
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The FLUSHOUTPUT function flushes the output buffer.

Parameter	Description
<i>pbCommand</i>	Points to a variable that contains a reserved value. It must be zero.
<i>hDevice</i>	Identifies the device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

FLUSHINPUT

■ **int DosDevIOctl(OL, *pbCommand*, 0x0045, 0x0008, *hDevice*)**
PBYTE *pbCommand*; pointer to buffer with command
HFILE *hDevice*; device handle

The FORMATVERIFY function formats and verifies a track on a drive. The function formats and verifies the specified track 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 to do the formatting.

Parameter	Description
<i>pbCommand</i>	Points to the structure that contains information about the format operation. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbCommand* parameter has the following form:

```
struct {
    BYTE bCommand;
    USHORT head;
    USHORT cylinder;
    USHORT reserved;
    USHORT cSectors;
    struct {
        BYTE bCylinder;
        BYTE bHead;
        BYTE idSector;
        BYTE bBytesSector;
    } FormatTable[N];
};
```

Field	Description
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.

head Specifies the number of the physical head on which to perform the operation.

cylinder Specifies the cylinder number for the operation.

cSectors Specifies the number of sectors on the track being formatted.

FormatTable[N]

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.

Comments

Some controllers do not support formatting tracks with varying sector sizes, so in general the program must make sure that the sector sizes specified in the format table are all the same.

■ **USHORT DosDevIOCtl**(*pusBaudRate*, 0L, 0x0061, 0x0001, *hDevice*)
PUSHORT *pusBaudRate*; pointer to variable for baud rate
HFILE *hDevice*; device handle

The GETBAUDRATE function retrieves the baud (bit) rate for the specified serial device. The baud rate specifies the number of bits per second that the serial device transmits or receives.

Parameter	Description
<i>pusBaudRate</i>	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 previously created by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

See Also

SETBAUDRATE

- **int DosDevIOCtl**(*pusCount*, 0L, 0x0060, 0x0007, *hDevice*)
PUSHORT *pusCount*; pointer to variable for button count
HFILE *hDevice*; device handle

The GETBUTTONCOUNT function retrieves a count of the number of buttons on the mouse.

Parameter	Description
<i>pusCount</i>	Points to the variable that receives the count of buttons on the mouse.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

- **USHORT DosDevIOCtl**(*pfCommErr*, 0L, 0x006D, 0x0001, *hDevice*)
PUSHORT *pfCommErr*; pointer to variable to receive error
HFILE *hDevice*; device handle

The GETCOMMERROR function retrieves the communication error word. After copying the error-word value to the variable pointed to by the *pfCommErr* parameter, the function clears the error word.

GETCOMMERROR

Parameter	Description										
<i>pfCommErr</i>	Points to the variable that receives the communication status of the device. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.</td></tr><tr><td>0x0002</td><td>Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.</td></tr><tr><td>0x0004</td><td>The hardware detected a parity error.</td></tr><tr><td>0x0008</td><td>The hardware detected a framing error.</td></tr></table>	Value	Meaning	0x0001	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.	0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.	0x0004	The hardware detected a parity error.	0x0008	The hardware detected a framing error.
Value	Meaning										
0x0001	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.										
0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.										
0x0004	The hardware detected a parity error.										
0x0008	The hardware detected a framing error.										
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.										

Return Value

The return value is zero if the function is successful. When an error occurs, the function returns a “general failure” error code (ERROR_GEN_FAILURE), 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 SETDCBINFO function (0x0001,0x0053).

See Also

GETCOMMEVENT, GETCOMMSTATUS

- **USHORT** **DosDevIOCtl**(*pfEvent*, **0L**, **0x0072**, **0x0001**, *hDevice*)
PUSHORT *pfEvent*; pointer to variable for events
HFILE *hDevice*; device handle

The GETCOMMEVENT function retrieves the communications event flags from the internally maintained event word. After the function copies the

event flags to the variable pointed to by the *pfEvent* parameter, it clears the event word.

Parameter	Description																		
<i>pfEvent</i>	Points to the variable that receives the event flags. It can be a combination of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>Specifies that a character has been read from the serial-device receive hardware and placed in the receive queue.</td></tr> <tr> <td>0x0004</td><td>Specifies that 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.</td></tr> <tr> <td>0x0008</td><td>Specifies the clear-to-send (CTS) signal has changed state.</td></tr> <tr> <td>0x0010</td><td>Specifies the data-set-ready (DSR) signal has changed state.</td></tr> <tr> <td>0x0020</td><td>Specifies the data-carrier-detect (DCD) signal has changed state.</td></tr> <tr> <td>0x0040</td><td>Specifies a break has been detected.</td></tr> <tr> <td>0x0080</td><td>Specifies a parity, framing, or overrun error has occurred. An overrun can be a receive hardware overrun or a receive queue overrun.</td></tr> <tr> <td>0x0100</td><td>Specifies the trailing edge of the ring indicator (RI) has been detected.</td></tr> </table>	Value	Meaning	0x0001	Specifies that a character has been read from the serial-device receive hardware and placed in the receive queue.	0x0004	Specifies that 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.	0x0008	Specifies the clear-to-send (CTS) signal has changed state.	0x0010	Specifies the data-set-ready (DSR) signal has changed state.	0x0020	Specifies the data-carrier-detect (DCD) signal has changed state.	0x0040	Specifies a break has been detected.	0x0080	Specifies a parity, framing, or overrun error has occurred. An overrun can be a receive hardware overrun or a receive queue overrun.	0x0100	Specifies the trailing edge of the ring indicator (RI) has been detected.
Value	Meaning																		
0x0001	Specifies that a character has been read from the serial-device receive hardware and placed in the receive queue.																		
0x0004	Specifies that 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.																		
0x0008	Specifies the clear-to-send (CTS) signal has changed state.																		
0x0010	Specifies the data-set-ready (DSR) signal has changed state.																		
0x0020	Specifies the data-carrier-detect (DCD) signal has changed state.																		
0x0040	Specifies a break has been detected.																		
0x0080	Specifies a parity, framing, or overrun error has occurred. An overrun can be a receive hardware overrun or a receive queue overrun.																		
0x0100	Specifies the trailing edge of the ring indicator (RI) has been detected.																		
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.																		

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

0x40	Character is waiting to transmit immediately. For a full description, see the TRANSMITIMM function (0x0001,0x0044).
0x80	Receive is waiting for the data-set-ready (DSR) signal to be turned on. For a full description, see the SETDCB function (0x0001,0x0053).

hDevice Identifies the serial device that receives the device-control function. The handle must have been previously created by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when 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 (e.g., enabled for output handshaking for the modem-control signal) for the status to reflect that the device driver would be waiting for the given condition to transmit.

For example, 0x01 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 0x10 if the device driver is enabled for automatic transmit flow control (XON/XOFF) or the STOP-TRANSMIT function (0x0001,0x0047) has been used to tell the device driver to behave as if an XOFF character is received. The 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 0x08 if the device driver is enabled for automatic receive flow control. When in this state, the TRANSMITIMM function (0x0001,0x0044) can still be used to transmit characters immediately and the device driver can still automatically transmit XON characters.

See Also

GETCOMMEVENT

- **USHORT** **DosDevIOCtl**(*pusDCB*, 0L, 0x0073, 0x0001, *hDevice*)
PUSHORT *pusDCB*; pointer to buffer for device-control information
HFILE *hDevice*; device handle

The GETDCBINFO function retrieves device-control block information.

Parameter	Description
<i>pusDCB</i>	Points to the structure that receives the device-control block information. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. When an error occurs, the function returns a "general failure" error code (ERROR_GEN_FAILURE) and any data copied to the structure pointed to by the *pusDCB* parameter is not valid.

Structures

The structure pointed to by the *pusDCB* parameter has the following form:

```
struct {  
    USHORT usWriteTimeout;  
    USHORT usReadTimeout;  
    BYTE    bFlags1  
    BYTE    bFlags2  
    BYTE    bFlags3  
    BYTE    bErrorReplacementChar;  
    BYTE    bBreakReplacementChar;  
    BYTE    bXONChar;  
    BYTE    bXOFFChar;  
};
```

Field	Description
usWriteTimeout	Specifies the time-out in milliseconds.
usReadTimeout	Specifies the time-out in milliseconds.

bFlags1

Specifies the control and handshaking modes for the DTR and other signals. It can be a combination of the following values:

Value	Meaning
0x00	Disable data-terminal-ready (DTR) control mode.
0x01	Enable data-terminal-ready (DTR) control mode.
0x02	Enable data-terminal-ready (DTR) input handshaking.
0x04	Enable output handshaking using a clear-to-send (CTS) signal.
0x10	Enable output handshaking using a data-set-ready (DSR) signal.
0x20	Enable output handshaking using a data-carrier-detect (DCD) signal.
0x40	Enable input sensitivity using a data-set-ready (DSR) signal.

bFlags2

Specifies the flow control and replacement character modes. It can be a combination of the following values:

Value	Meaning
0x01	Enable automatic transmit flow control (XON/XOFF).
0x02	Enable automatic receive flow control (XON/XOFF).
0x04	Enable error replacement character.
0x08	Enable null stripping (remove null bytes).
0x10	Enable break replacement character.
0x40	Enable ready-to-send (RTS) control mode.
0x80	Enable ready-to-send (RTS) input handshaking.
0xC0	Enable toggling on transmit.

bFlags3

Specifies the time-out processing for the device. It can be a combination of the following values:

Value	Meaning
0x01	Enable write infinite time-out processing.
0x02	Enable normal read time-out processing.
0x04	Enable wait-for-something read time-out processing.
0x06	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.

Comments

A complete description of the **bFlags1**, **bFlags2**, and **bFlags3** fields is given with the SETDCBINFO function (0x0001,0x0053). Bits within these undefined fields receive the correct value so that the retrieved field can be used in future releases without unwittingly modifying the device-driver modes. In this case, the correct bit value is zero, but the program should not assume that zero will be used in all cases.

To ensure that only valid values are set in the device-control block, the program should call the GETDCBINFO function to fill the block, modify the settings, and then call the SETDCBINFO function with the modified block.

See Also

SETDCBINFO

- **int DosDevIOCtl(*pbBPB*, *pbCommand*, 0x0063, 0x0008, *hDevice*)**
PBYTE *pbBPB*; pointer to buffer for BPB
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The 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 is the current BPB that corresponds to the media currently in the disk drive. The other block is a recommended BPB, based on the type of media that corresponds to the physical device. For example, a high-density disk drive has a BPB for a 96-tpi (tracks-per-inch) floppy disk; a low-density disk drive has a BPB for a 48-tpi floppy disk.

Parameter	Description
<i>pbBPB</i>	Points to the structure that receives the BPB. For a full description, see the following "Structures" section.
<i>pbCommand</i>	Points to the variable specifying which BPB to retrieve. If the variable is 0x0000, the function retrieves the recommended BPB for the drive. The recommended BPB for the drive is the BPB for the physical device. If the variable is 0x0001, the function retrieves the BPB for the media currently in the drive.
<i>hDevice</i>	Identifies the disk drive that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbBPB* parameter has the following form:

```
struct {
    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;
};
```

Field	Description																		
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.																		
cCylinders	Specifies the number of cylinders defined for the physical device.																		
bDeviceType	Specifies the physical layout of the specified device. It can be one of the following values:																		
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>48 tracks-per-inch, low-density floppy disk drive</td></tr> <tr> <td>0x0001</td><td>96 tracks-per-inch, high-density floppy disk drive</td></tr> <tr> <td>0x0002</td><td>3.5-inch (720K) floppy disk drive</td></tr> <tr> <td>0x0003</td><td>8-inch, single-density floppy disk drive</td></tr> <tr> <td>0x0004</td><td>8-inch, double-density floppy disk drive</td></tr> <tr> <td>0x0005</td><td>Hard disk</td></tr> <tr> <td>0x0006</td><td>Tape drive</td></tr> <tr> <td>0x0007</td><td>Other (unknown type of device)</td></tr> </table>	Value	Meaning	0x0000	48 tracks-per-inch, low-density floppy disk drive	0x0001	96 tracks-per-inch, high-density floppy disk drive	0x0002	3.5-inch (720K) floppy disk drive	0x0003	8-inch, single-density floppy disk drive	0x0004	8-inch, double-density floppy disk drive	0x0005	Hard disk	0x0006	Tape drive	0x0007	Other (unknown type of device)
Value	Meaning																		
0x0000	48 tracks-per-inch, low-density floppy disk drive																		
0x0001	96 tracks-per-inch, high-density floppy disk drive																		
0x0002	3.5-inch (720K) floppy disk drive																		
0x0003	8-inch, single-density floppy disk drive																		
0x0004	8-inch, double-density floppy disk drive																		
0x0005	Hard disk																		
0x0006	Tape drive																		
0x0007	Other (unknown type of device)																		

fDeviceAttr Specifies information about the drive. It can be one or both of the following values:

Value	Meaning
0x0001	Removable media.
0x0002	The media can detect changes.

See Also

SETDEVICEPARAMS

■ **int DosDevIOCtl(*pfEvents*, 0L, 0x0065, 0x0007, *hDevice*)**
PUSHORT *pfEvents*; pointer to variable for event mask
HFILE *hDevice*; device handle

The GETEVENTMASK function retrieves the event mask of the current pointing device.

Parameter	Description																
<i>pfEvents</i>	Points to the variable that receives the event mask. The event mask can be any combination of the following values:																
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>All mouse motion, no buttons pressed.</td></tr> <tr> <td>0x0002</td><td>Motion with button 1 pressed.</td></tr> <tr> <td>0x0004</td><td>Button 1 pressed.</td></tr> <tr> <td>0x0008</td><td>Motion with button 2 pressed.</td></tr> <tr> <td>0x0010</td><td>Button 2 pressed.</td></tr> <tr> <td>0x0020</td><td>Motion with button 3 pressed.</td></tr> <tr> <td>0x0040</td><td>Button 3 pressed.</td></tr> </table>	Value	Meaning	0x0001	All mouse motion, no buttons pressed.	0x0002	Motion with button 1 pressed.	0x0004	Button 1 pressed.	0x0008	Motion with button 2 pressed.	0x0010	Button 2 pressed.	0x0020	Motion with button 3 pressed.	0x0040	Button 3 pressed.
Value	Meaning																
0x0001	All mouse motion, no buttons pressed.																
0x0002	Motion with button 1 pressed.																
0x0004	Button 1 pressed.																
0x0008	Motion with button 2 pressed.																
0x0010	Button 2 pressed.																
0x0020	Motion with button 3 pressed.																
0x0040	Button 3 pressed.																
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.																

Return Value

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

See Also

SETEVENTMASK

- **int DosDevIOCtl(*pbFrameCtl*, *pbCommand*, 0x0062, 0x0005, *hDevice*)**
PBYTE *pbFrameCtl*; pointer to buffer for frame settings
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The GETFRAMECTL function retrieves frame-control information for a print device.

Parameter	Description
<i>pbFrameCtl</i>	Points to the structure that receives the frame-control information. For a full description, see the following “Structures” section.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the printer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbFrameCtl* parameter has the following form:

```
struct {  
    BYTE bCharsPerLine;  
    BYTE bLinesPerInch;  
};
```

Field	Description
bCharsPerLine	Specifies the number of characters on a line, 80 or 132.
bLinesPerInch	Specifies the number of lines per inch, 6 or 8.

See Also

SETFRAMECTL

■ **int** DosDevIOCtl(*pfHotKey*, 0L, 0x0089, 0x0007, *hDevice*)
PUSHORT *pfHotKey*; pointer to variable for hot-key flag
HFILE *hDevice*; device handle

The GETHOTKEYBUTTON function retrieves the mouse button equivalent for the system hot key.

Parameter	Description										
<i>pfHotKey</i>	Points to the variable to receive the hot key button. It can be one or more of the following values:										
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>No system hot key desired.</td></tr> <tr> <td>MHK_BUTTON1</td><td>Button 1 is system hot key.</td></tr> <tr> <td>MHK_BUTTON2</td><td>Button 2 is system hot key.</td></tr> <tr> <td>MHK_BUTTON3</td><td>Button 3 is system hot key.</td></tr> </table>	Value	Meaning	0x0001	No system hot key desired.	MHK_BUTTON1	Button 1 is system hot key.	MHK_BUTTON2	Button 2 is system hot key.	MHK_BUTTON3	Button 3 is system hot key.
Value	Meaning										
0x0001	No system hot key desired.										
MHK_BUTTON1	Button 1 is system hot key.										
MHK_BUTTON2	Button 2 is system hot key.										
MHK_BUTTON3	Button 3 is system hot key.										
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.										

Return Value

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

Comments

If 0x0001 is specified, no system hot-key support is provided. If multiple values are given, not including 0x0001, the system hot key is interpreted as requiring the indicated buttons to be pressed simultaneously.

See Also

SETHOTKEYBUTTON

- int 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 GETINFINITERETRY function retrieves an infinite retry setting for a print device.

Parameter	Description
<i>pfRetry</i>	Points to the variable to receive the infinite retry setting. The variable is 0x0000 if infinite retry is disabled. It is 0x0001 if retry is enabled.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the printer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

SETINFINITERETRY

- int DosDevIOCtl**(*pbInputMode*, **0L**, **0x0071**, **0x0004**, *hDevice*)
PBYTE *pbInputMode*; pointer to variable for input mode
HFILE *hDevice*; device handle

The GETINPUTMODE function retrieves the input mode of the screen group of the currently 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.

Parameter	Description
<i>pbInputMode</i>	Points to the character variable to receive the input mode. If it is 0x00, the keyboard has cooked input mode. If it is 0x80, the keyboard has raw input mode.

hDevice Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the **DosOpen** function.

Return Value

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

See Also

SETINPUTMODE

■ **USHORT** **DosDevIOCtl**(*pcReceiveQue*, **0L**, **0x0068**, **0x0001**, *hDevice*)
PUSHORT *pcReceiveQue*; pointer to buffer for count
HFILE *hDevice*; device handle

The GETINQUEECOUNT function retrieves the number of characters in the receive queue.

Parameter	Description
<i>pcReceiveQue</i>	Points to a structure that receives the count of characters in the receive queue. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

Structures

The structure pointed to by the *pcReceiveQue* parameter has the following form:

```
struct {
    USHORT cbChars;
    USHORT cbQueue;
};
```

GETINQUECOUNT

Field	Description
cbChars	Specifies the number of received characters in the device-driver receive queue.
cbQueue	Specifies the size in bytes of the receive queue.

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

GETOUTQUECOUNT

■ **int** DosDevIOctl(*pfFlags*, **0L**, **0x0072**, **0x0004**, *hDevice*)
PBYTE *pfFlags*; pointer to variable for flags
HFILE *hDevice*; device handle

The GETINTERIMFLAG function retrieves interim character flags.

Parameter	Description
<i>pfFlags</i>	Points to the variable that receives interim flags. If the variable is 0x20, the program requested conversion. If it is 0x80, the interim console flag is on.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

SETINTERIMFLAG

```

int DosDevIOCtl(pbType, 0L, 0x0077, 0x0004, hDevice)
PBYTE pbType;           pointer to buffer for keyboard type
HFILE hDevice;          device handle

```

The `GETKEYBDTYPE` function retrieves information about the type of keyboard being used.

Parameter	Description
<i>pbType</i>	Points to the structure to receive the keyboard type. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbType* parameter has the following form:

```
struct {
    USHORT usType;
    USHORT reserved1;
    USHORT reserved2;
};
```

Field	Description
usType	Specifies the keyboard type. It can be one of the following values:

	Value	Meaning
	0x0000	IBM PC/AT keyboard.
	0x0001	IBM enhanced keyboard.
	Values from 0x0002 to 0x0007 are reserved for Japanese keyboards.	
reserved1	Specifies a reserved value. It is set to zero.	
reserved2	Specifies a reserved value. It is set to zero.	

- | | | |
|---------------|---------------------|--|
| USHORT | DosDevIOCtrl | (<i>pbLineCtrl</i> , 0L , 0x0062 , 0x0001 , <i>hDevice</i>) |
| PBYTE | <i>pbLineCtrl</i> ; | pointer to buffer for control settings |
| HFILE | <i>hDevice</i> ; | device handle |

The GETLINECTRL function retrieves the line characteristics (stop bits, parity, data bits, break) for the specified device.

Parameter	Description
<i>pbLineCtrl</i>	Points to a structure that receives the settings for the number of data bits, parity, and number of stop bits. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (`ERROR_GEN_FAILURE`) when an error occurs.

Structures

The structure pointed to by the *pbLineCtrl* parameter has the following form:

```
struct {
    BYTE bDataBits;
    BYTE bParity;
    BYTE bStopBits;
    BYTE fbTransBreak;
};
```

Field	Description												
bDataBits	Specifies the number of data bits to be used. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x05</td><td>5 data bits</td></tr><tr><td>0x06</td><td>6 data bits</td></tr><tr><td>0x07</td><td>7 data bits</td></tr><tr><td>0x08</td><td>8 data bits</td></tr></table>	Value	Meaning	0x05	5 data bits	0x06	6 data bits	0x07	7 data bits	0x08	8 data bits		
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: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x00</td><td>No parity</td></tr><tr><td>0x01</td><td>Odd parity</td></tr><tr><td>0x02</td><td>Even parity</td></tr><tr><td>0x03</td><td>Mark parity (parity bit is always 1)</td></tr><tr><td>0x04</td><td>Space parity (parity bit is always 0)</td></tr></table>	Value	Meaning	0x00	No parity	0x01	Odd parity	0x02	Even parity	0x03	Mark parity (parity bit is always 1)	0x04	Space parity (parity bit is always 0)
Value	Meaning												
0x00	No parity												
0x01	Odd parity												
0x02	Even parity												
0x03	Mark parity (parity bit is always 1)												
0x04	Space parity (parity bit is always 0)												
bStopBits	Specifies the number of stop bits used. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x00</td><td>1 stop bit</td></tr><tr><td>0x01</td><td>1.5 stop bits (valid with 5-bit word length only)</td></tr><tr><td>0x02</td><td>2 stop bits (not valid with 5-bit word length)</td></tr></table>	Value	Meaning	0x00	1 stop bit	0x01	1.5 stop bits (valid with 5-bit word length only)	0x02	2 stop bits (not valid with 5-bit word length)				
Value	Meaning												
0x00	1 stop bit												
0x01	1.5 stop bits (valid with 5-bit word length only)												
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 is not being transmitted. If it is 0x01, a break is being transmitted.												

See Also

SETLINECTRL

GETLINESTATUS

- **USHORT** **DosDevIOctl**(*pbTransStatus*, **0L**, **0x0065**, **0x0001**, *hDevice*)
PBYTE *pbTransStatus*; pointer to variable for status
HFILE *hDevice*; device handle

The GETLINESTATUS function retrieves the transmit data status for the specified serial device.

Parameter	Description														
<i>pbTransStatus</i>	Points to the variable that receives the transmit data status. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x01</td><td>Write request packets in progress or queued.</td></tr><tr><td>0x02</td><td>Data in the device-driver transmit queue.</td></tr><tr><td>0x04</td><td>Transmit hardware currently transmitting data.</td></tr><tr><td>0x08</td><td>Character waiting to be transmitted "immediately."</td></tr><tr><td>0x10</td><td>Waiting to automatically transmit XON.</td></tr><tr><td>0x20</td><td>Waiting to automatically transmit XOFF.</td></tr></table>	Value	Meaning	0x01	Write request packets in progress or queued.	0x02	Data in the device-driver transmit queue.	0x04	Transmit hardware currently transmitting data.	0x08	Character waiting to be transmitted "immediately."	0x10	Waiting to automatically transmit XON.	0x20	Waiting to automatically transmit XOFF.
Value	Meaning														
0x01	Write request packets in progress or queued.														
0x02	Data in the device-driver transmit queue.														
0x04	Transmit hardware currently transmitting data.														
0x08	Character waiting to be transmitted "immediately."														
0x10	Waiting to automatically transmit XON.														
0x20	Waiting to automatically transmit XOFF.														
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.														

Return Value

The return value is zero if the function is successful. The function returns a "general failure" error code (**ERROR_GEN_FAILURE**) when an error occurs.

See Also

GETCOMMSTATUS

- **int** **DosDevIOctl**(*pbDrive*, *pbCommand*, **0x0021**, **0x0008**, *hDevice*)
PBYTE *pbDrive*; pointer to variable for drive
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The GETLOGICALMAP function retrieves the mapping of a logical drive.

Parameter	Description
<i>pbDrive</i>	Points to the variable to receive the logical drive number. The number 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 this physical drive.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the physical device to receive the device-control function. The handle must have been previously created using the DosOpen function.

Return Value

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

See Also

SETLOGICALDRIVE

■ **int** DosDevIOCtl(*pcMickey*s, 0L, 0x0061, 0x0007, *hDevice*)
PUSHORT *pcMickey*s; pointer to variable for mickeys
HFILE *hDevice*; device handle

The GETMICKEYCOUNT function retrieves the count of the number of mickeys per centimeter for a given mouse device.

Parameter	Description
<i>pcMickey</i> s	Points to the variable to receive the number of mickeys per centimeter. The number can be any value from 0 to 32,767.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

- **USHORT** **DosDevIOCtl**(*pbCtrlSignals*, **0L**, **0x0067**, **0x0001**, *hDevice*)
PBYTE *pbCtrlSignals*; pointer to variable for control signals
HFILE *hDevice*; device handle

The GETMODEMINPUT function retrieves the modem-control input signals for the specified device.

Parameter	Description										
<i>pbCtrlSignals</i>	Points to the variable that receives the modem-control signals. It can be a combination of the following values:										
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x10</td><td>Clear-to-send (CTS) signal is on. If not given, the signal is off.</td></tr> <tr> <td>0x20</td><td>Data-set-ready (DSR) signal is on. If not given, the signal is off.</td></tr> <tr> <td>0x40</td><td>Ring-indicator (RI) signal is on. If not given, the signal is off.</td></tr> <tr> <td>0x80</td><td>Data-carrier-detect (DCD) signal is on. If not given, the signal is off.</td></tr> </table>	Value	Meaning	0x10	Clear-to-send (CTS) signal is on. If not given, the signal is off.	0x20	Data-set-ready (DSR) signal is on. If not given, the signal is off.	0x40	Ring-indicator (RI) signal is on. If not given, the signal is off.	0x80	Data-carrier-detect (DCD) signal is on. If not given, the signal is off.
Value	Meaning										
0x10	Clear-to-send (CTS) signal is on. If not given, the signal is off.										
0x20	Data-set-ready (DSR) signal is on. If not given, the signal is off.										
0x40	Ring-indicator (RI) signal is on. If not given, the signal is off.										
0x80	Data-carrier-detect (DCD) signal is on. If not given, the signal is off.										
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.										

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

See Also

GETMODEMOUTPUT, SETMODEMCTRL

- **USHORT** **DosDevIOCtl**(*pbCtrlSignals*, **0L**, **0x0066**, **0x0001**, *hDevice*)
PBYTE *pbCtrlSignals*; pointer to variable for control signals
HFILE *hDevice*; device handle

The GETMODEMOUTPUT function retrieves the modem-control output signals for the specified device.

Parameter	Description						
<i>pbCtrlSignals</i>	Points to the variable that receives the modem-control signals. It can be a combination of the following values:						
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x01</td><td>Data-terminal-ready (DTR) signal is on. If not given, the signal is off.</td></tr> <tr> <td>0x02</td><td>Request-to-send (RTS) signal is on. If not given, the signal is off.</td></tr> </table>	Value	Meaning	0x01	Data-terminal-ready (DTR) signal is on. If not given, the signal is off.	0x02	Request-to-send (RTS) signal is on. If not given, the signal is off.
Value	Meaning						
0x01	Data-terminal-ready (DTR) signal is on. If not given, the signal is off.						
0x02	Request-to-send (RTS) signal is on. If not given, the signal is off.						
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.						

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

See Also

GETMODEMINPUT, SETMODEMCTRL

■ **int** DosDevIOCtl(*pfStatus*, **0L**, **0x0062**, **0x0007**, *hDevice*)
PUSHORT *pfStatus*; pointer to variable status
HFILE *hDevice*; device handle

The GETMOUSTATUS function retrieves the current status flags of the mouse device driver.

Parameter	Description								
<i>pfStatus</i>	Points to the variable to receive the status flags. It can be any combination of the following values:								
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>Event queue is busy with I/O.</td></tr> <tr> <td>0x0002</td><td>Block read is in progress.</td></tr> <tr> <td>0x0004</td><td>Flush in progress.</td></tr> </table>	Value	Meaning	0x0001	Event queue is busy with I/O.	0x0002	Block read is in progress.	0x0004	Flush in progress.
Value	Meaning								
0x0001	Event queue is busy with I/O.								
0x0002	Block read is in progress.								
0x0004	Flush in progress.								

GETMOUSTATUS

	0x0008	Pointer-draw routine is disabled (device in unsupported mode).
	0x0100	Interrupt-level pointer-draw routine is not called.
	0x0200	Mouse data is being returned in mickeys (not pixels).
<i>hDevice</i>		Identifies the mouse device to receive the device-control function. The handle must have been previously created using the DosOpen function.

Return Value

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

See Also

SETMOUSTATUS

- **USHORT** **DosDevIOCtl**(*pcTransmitQue*, **0L**, **0x0069**, **0x0001**, *hDevice*)
PUSHORT *pcTransmitQue*; pointer to buffer for count
HFILE *hDevice*; device handle

The GETOUTQUEECOUNT function retrieves a count of the characters in the transmit queue.

Parameter	Description
<i>pcTransmitQue</i>	Points to a structure that receives the count of characters in the transmit queue. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. The function returns a "general failure" error code (ERROR_GEN_FAILURE) when an error occurs.

Structures

The structure pointed to by the *pcTransmitQue* parameter has the following form:

```
struct {
    USHORT cbChars;
    USHORT cbQueue;
};
```

Field	Description
cbChars	Specifies the number of characters to transmit in the device-driver transmit queue.
cbQueue	Specifies the size in bytes of the transmit queue.

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

GETINQUEECOUNT

■ **int DosDevIOCtl(*pbBlock*, *pbCommand*, 0x0063, 0x0009, *hDevice*)**
PBYTE *pbBlock*; physical disk block
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The GETPHYSDEVICEPARAMS function retrieves the device parameters for a physical device. The retrieved parameters apply to the entire physical disk.

GETPHYSDEVICEPARAMS

Parameter	Description
<i>pbBlock</i>	Points to the structure to receive the device parameters. For a full description, see the “Structures” section.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the physical device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbBlock* parameter has the following form:

```
struct {
    USHORT reserved1;
    USHORT cCylinders;
    USHORT cHeads;
    USHORT cSectorsPerTrack;
    USHORT reserved2;
    USHORT reserved3;
    USHORT reserved4;
    USHORT reserved5;
};
```

Field	Description
reserved1	Specifies a reserved value. It 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.

- **int 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 GETPRINTERSTATUS function retrieves the status of a print device.

Parameter	Description														
<i>pfStatus</i>	Points to the variable to receive the printer status. It can be a combination of the following values:														
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>Timeout occurred.</td></tr> <tr> <td>0x0008</td><td>I/O error occurred.</td></tr> <tr> <td>0x0010</td><td>Printer selected.</td></tr> <tr> <td>0x0020</td><td>Printer out of paper.</td></tr> <tr> <td>0x0040</td><td>Printer acknowledged.</td></tr> <tr> <td>0x0080</td><td>Printer not busy.</td></tr> </table>	Value	Meaning	0x0001	Timeout occurred.	0x0008	I/O error occurred.	0x0010	Printer selected.	0x0020	Printer out of paper.	0x0040	Printer acknowledged.	0x0080	Printer not busy.
Value	Meaning														
0x0001	Timeout occurred.														
0x0008	I/O error occurred.														
0x0010	Printer selected.														
0x0020	Printer out of paper.														
0x0040	Printer acknowledged.														
0x0080	Printer not busy.														
<i>pbCommand</i>	Points to the variable containing a reserved value. The value must be zero.														
<i>hDevice</i>	Identifies the printer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.														

Return Value

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

■ **int DosDevIOCtl(*pbFunctionInfo*, 0L, 0x0072, 0x0003, *hDevice*)**
PBYTE *pbFunctionInfo*; pointer to buffer for function
HFILE *hDevice*; device handle

The GETPTRDRAWADDRESS function retrieves the entry point address and other information for the pointer-draw function. The pointer-draw function draws the mouse pointer on the screen.

Parameter	Description
<i>pbFunctionInfo</i>	Points to the structure to receive the function information. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the pointer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbFunctionInfo* parameter has the following form:

```
struct {  
    USHORT usReturnCode;  
    PEN     pfnDraw;  
    SEL     selData;  
};
```

Field	Description
usReturnCode	Specifies the return code of the function.
pfnDraw	Specifies the entry point address of the pointer-draw function.
selData	Specifies the data segment selector identifying the data segment used by the pointer-draw function.

Comments

The pointer-draw function is used by the mouse device driver to update the pointer image on the screen. The mouse driver retrieves the address and saves it to be used whenever the pointer moves.

- **int** DosDevIOCtl(*pplPosition*, **0L**, **0x0067**, **0x0007**, *hDevice*)
PPTRLOC *pplPosition*; pointer to buffer for position
HFILE *hDevice*; device handle

The GETPTRPOS function retrieves the position of the current screen's pointer image.

Parameter	Description
<i>pplPosition</i>	Points to the structure to receive the new pointer position. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pplPosition* parameter has the following format:

```
typedef struct _PTRLOC {
    USHORT row;
    USHORT col;
} PTRLOC;
```

Field	Description
row	Specifies the <i>x</i> -coordinate of the pointer.
col	Specifies the <i>y</i> -coordinate of the pointer.

Comments

The coordinate values depend on the display mode. If the display is in text mode, character position values are given. But if the display is in graphics mode, pixel values are given.

See Also

SETPTRPOS

- **int** DosDevIOCtl(*pbBuffer*, *ppsShape*, **0x0068**, **0x0007**, *hDevice*)
PBYTE *pbBuffer*; pointer to buffer for pointer masks
PPTRSHAPE *ppsShape*; pointer to buffer for shape information
HFILE *hDevice*; device handle

The GETPTRSHAPE function retrieves the current mouse pointer image.

Parameter	Description
<i>pbBuffer</i>	Points to the buffer to receive 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.
<i>ppsShape</i>	Points to the structure to receive the pointer information and shape. For a full description, see the following "Structures" section.

hDevice Identifies the mouse device to receive the device-control function. The handle must have been previously created using the **DosOpen** function.

Return Value

The function will exit in a normal state if the input pointer-image buffer length is greater than or equal to the amount of storage required for the pointer-image. Also, the current pointer information will be returned in the pointer-data record, and the pointer-image data will be copied into the data-packet buffer.

An “invalid buffer size” error will occur if the length of the input pointer-image buffer is smaller than the amount of storage necessary to perform the data copy. Also, the buffer length that is returned will be the minimum value.

Structures

The structure pointed to by the *ppsShape* parameter has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

Field	Description
cb	Specifies the length in bytes of the pointer-shape buffer.
col	Specifies the width in columns of the pointer image.
row	Specifies the height in rows of the pointer image.
colHot	Specifies the offset in columns from the first columns to the pointer hot spot.
rowHot	Specifies the offset in rows from the first row to the pointer hot spot.

Comments

The parameter values are in the same mode as the current screen group display mode. For text mode, these values are character values; for graphics mode, they are pixel values.

On input, the only field in the pointer definition record that is used by the mouse device driver is the length of pointer-image buffer. This buffer is pointed to by the *pbBuffer* parameter.

See Also

SETPTRSHAPE

■ **int** DosDevIOCtl(*pmqiStatus*, 0L, 0x0064, 0x0007, *hDevice*)
PMOQUEINFO *pmqiStatus*; pointer to buffer for queue status
HFILE *hDevice*; device handle

The GETQUESTATUS function retrieves both the current number of queued elements in the event queue, and the maximum number of elements allowed in an event queue.

Parameter	Description
<i>pmqiStatus</i>	Points to the structure to receive the queue status. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pmqiStatus* parameter has the following form:

```
typedef struct _MOQUEINFO {
    USHORT cEvents;
    USHORT cmaxEvents;
} MOQUEINFO;
```

Field	Description
cEvents	Specifies the number of event-queue elements in the queue.
cmaxEvents	Specifies the maximum number of elements that can be in the queue.

- int** **DosDevIOCtl**(*psfFactors*, **0L**, **0x0066**, **0x0007**, *hDevice*)
PSCALEFACT *psfFactors*; pointer to buffer for scaling factors
HFILE *hDevice*; device handle

The 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 mouse event. In graphics mode, this ratio is in mickeys-per-pixel. In text mode, this ratio is in mickeys-per-character. The ratio values default to one mickey-per-row unit and one mickey-per-column unit.

Parameter	Description
<i>psfFactors</i>	Points to the structure to receive the scale factors. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *psfFactors* parameter has the following form:

```
typedef struct _SCALEFACT {
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

Field	Description
rowScale	Specifies the horizontal coordinate scaling factor. It must be a value from 0 to 32,767.
colScale	Specifies the vertical coordinate scaling factor. It must be a value from 0 to 32,767.

See Also

SETSCALEFACTORS

■

```
int DosDevIOctl(pbHotKeyBuf, pcHotKeys, 0x0076, 0x0004, hDevice)
PBYTE pbHotKeyBuf;           pointer to variable for hot keys
PUSHORT pcHotKeys;           pointer to variable for hot-key count
HFILE hDevice;               device handle
```

The GETSESMGRHOTKEY function retrieves the hot-key information structures for the currently defined hot keys. If the variable pointed to by the *pcHotKeys* parameter is 0x0000, the function returns the number of currently defined hot keys. The program can use this number to allocate sufficient space to retrieve the actual hot-key information (retrieved by setting the variable to 0x0001).

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.

Parameter	Description
<i>pbHotKeyBuf</i>	Points to the buffer to receive hot-key information structures. The size of the buffer must be at least as large as the number of structures requested. For a full description, see the following “Structures” section.
<i>pcHotKeys</i>	Points to the variable specifying the number of hot-key information structures to retrieve. If the variable is 0x0000, the function copies a value to the variable that specifies the maximum number of hot keys the keyboard device driver can support. If it is 0x0001, the function copies a value to this variable specifying the actual number of hot keys currently supported. The function also copies the hot-key information to the buffer pointed to by the <i>pbHotKeyBuf</i> parameter.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structures pointed to by the *pbHotKeyBuf* parameter has the following form:

```
struct {
    USHORT fHotKey;
    UCHAR  scancodeMake;
    UCHAR  scancodeBreak;
    USHORT idHotKey;
};
```

Field	Description																						
fHotKey	Specifies the setting for the session manager hot key. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>Right SHIFT key down.</td></tr><tr><td>0x0002</td><td>Left SHIFT key down.</td></tr><tr><td>0x0100</td><td>Left CONTROL key down.</td></tr><tr><td>0x0200</td><td>Left ALT key down.</td></tr><tr><td>0x0400</td><td>Right CONTROL key down.</td></tr><tr><td>0x0800</td><td>Right ALT key down.</td></tr><tr><td>0x1000</td><td>SCROLL LOCK key down.</td></tr><tr><td>0x2000</td><td>NUMLOCK key down.</td></tr><tr><td>0x4000</td><td>CAPSLOCK key down.</td></tr><tr><td>0x8000</td><td>SYSREQ key down.</td></tr></table>	Value	Meaning	0x0001	Right SHIFT key down.	0x0002	Left SHIFT key down.	0x0100	Left CONTROL key down.	0x0200	Left ALT key down.	0x0400	Right CONTROL key down.	0x0800	Right ALT key down.	0x1000	SCROLL LOCK key down.	0x2000	NUMLOCK key down.	0x4000	CAPSLOCK key down.	0x8000	SYSREQ key down.
Value	Meaning																						
0x0001	Right SHIFT key down.																						
0x0002	Left SHIFT key down.																						
0x0100	Left CONTROL key down.																						
0x0200	Left ALT key down.																						
0x0400	Right CONTROL key down.																						
0x0800	Right ALT key down.																						
0x1000	SCROLL LOCK key down.																						
0x2000	NUMLOCK key down.																						
0x4000	CAPSLOCK key down.																						
0x8000	SYSREQ key down.																						
scancodeMake	Specifies the scan code of the hot-key make. If given, the system detects the hot key when the user presses the key that generates this scan code.																						
scancodeBreak	Specifies the scan code of the hot-key break. If 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 is a value from 0 to 15.																						

The **scancodeMake** and **scancodeBreak** fields are mutually exclusive; either may be specified, but not both. The use of either field indicates when to recognize the hot key.

See Also

SETSESMGRHOTKEY

■

```
int DosDevIOCtl(pbShiftState, 0L, 0x0073, 0x0004, hDevice)
PBYTE pbShiftState;           pointer to buffer for shift state
HFILE hDevice;                 device handle
```

The GETSHIFTSTATE function retrieves the shift state of the screen group for the currently active process.

Parameter	Description
<i>pbShiftState</i>	Points to the structure to receive the shift state. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbShiftState* parameter has the following form:

```
struct {
    USHORT fState;
    BYTE   fNLS;
};
```

Field	Description										
fState	Specifies the state of the shift keys:										
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>Right SHIFT key down.</td></tr><tr><td>0x0002</td><td>Left SHIFT key down.</td></tr><tr><td>0x0004</td><td>Either CONTROL key down.</td></tr><tr><td>0x0008</td><td>Either ALT key down.</td></tr></table>	Value	Meaning	0x0001	Right SHIFT key down.	0x0002	Left SHIFT key down.	0x0004	Either CONTROL key down.	0x0008	Either ALT key down.
Value	Meaning										
0x0001	Right SHIFT key down.										
0x0002	Left SHIFT key down.										
0x0004	Either CONTROL key down.										
0x0008	Either ALT key down.										

GETSHIFTSTATE

0x0010	SCROLL LOCK mode on.
0x0020	NUMLOCK mode on.
0x0040	CAPSLOCK mode on.
0x0080	INSERT mode on.
0x0100	Left CONTROL key down.
0x0200	Left ALT key down.
0x0400	Right CONTROL key down.
0x0800	Right ALT key down.
0x1000	SCROLL LOCK key down.
0x2000	NUMLOCK key down.
0x4000	CAPSLOCK key down.
0x8000	SYSREQ key down.

fNLS Specifies the state of the national-language-support keys.

Comments

The shift state is set by incoming keystrokes. It can also be set by using the SETSHIFTSTATE function (0x0004,0x0053).

See Also

SETSHIFTSTATE

■ **int DosDevIOCtl(OL, *pbCommand*, 0x0046, 0x0005, *hDevice*)**
PBYTE *pbCommand*; command value
HFILE *hDevice*; device handle

The INITPRINTER function initializes a print device.

Parameter	Description
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the printer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

■ **int DosDevIOCtl(0L, *pbCommand*, 0x0000, 0x0008, *hDevice*)**
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The LOCKDRIVE function locks a disk drive. Locking a disk drive prevents file input/output (I/O) by another process on the volume in the disk drive. The function succeeds only if there is one file handle open on the volume in the disk drive. This is necessary since the desired result is to exclude all other I/O to the volume.

Parameter	Description
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the disk drive that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

UNLOCKDRIVE

■ **int DosDevIOCtl(0L, *pbCommand*, 0x0000, 0x0009, *hDevice*)**
PBYTE *pbCommand*; command information
HFILE *hDevice*; device handle

The LOCKPHYSDRIVE function locks the physical drive and any of its associated logical units. The function also affects the logical units on the physical drive.

Parameter	Description
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.

hDevice Identifies the disk-drive device to receive the device-control function. The handle must have been previously created using the **DosOpen** function.

Return Value

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

See Also

UNLOCKPHYSDRIVE

■ **int DosDevIOCtl**(*pkkiBuffer*, *pusStatus*, **0x0075**, **0x0004**, *hDevice*)
PKBDKEYINFO *pkkiBuffer*; pointer to buffer for keystroke
PUSHORT *pusStatus*; pointer to variable for status
HFILE *hDevice*; device handle

The PEEKCHAR function retrieves one character data record from the head of the keyboard input buffer of the screen group for the active process. The character data record is not removed from the keyboard input buffer.

Parameter	Description								
<i>pkkiBuffer</i>	Points to the buffer to receive the character data record. For a full description, see the following “Structures” section.								
<i>pusStatus</i>	Points to the variable to receive the keyboard status. It can be a combination of the following values:								
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>No character data record is available.</td></tr> <tr> <td>0x0001</td><td>Character data record is retrieved.</td></tr> <tr> <td>0x8000</td><td>Input mode is raw. If not given, input mode is cooked.</td></tr> </table>	Value	Meaning	0x0000	No character data record is available.	0x0001	Character data record is retrieved.	0x8000	Input mode is raw. If not given, input mode is cooked.
Value	Meaning								
0x0000	No character data record is available.								
0x0001	Character data record is retrieved.								
0x8000	Input mode is raw. If not given, input mode is cooked.								
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.								

Return Value

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

Structures

The structure pointed to by the *pkkiBuffer* parameter has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

Field	Description										
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: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>A shift key is received (valid only in raw mode when shift report is on).</td></tr> <tr> <td>0x0020</td><td>Conversion requested.</td></tr> <tr> <td>0x0040</td><td>Final character received.</td></tr> <tr> <td>0x0080</td><td>Interim character received.</td></tr> </table>	Value	Meaning	0x0001	A shift key is received (valid only in raw mode when shift report is on).	0x0020	Conversion requested.	0x0040	Final character received.	0x0080	Interim character received.
Value	Meaning										
0x0001	A shift key is received (valid only in raw mode when shift report is on).										
0x0020	Conversion requested.										
0x0040	Final character received.										
0x0080	Interim character received.										
bNlsShift	Specifies a reserved value. It must be zero.										
fsState	Specifies the state of the shift keys. It can be any combination of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0000</td><td>SHIFT key up.</td></tr> <tr> <td>0x0001</td><td>Right SHIFT key down.</td></tr> </table>	Value	Meaning	0x0000	SHIFT key up.	0x0001	Right SHIFT key down.				
Value	Meaning										
0x0000	SHIFT key up.										
0x0001	Right SHIFT key down.										

0x0002	Left SHIFT key down.
0x0004	Either CONTROL key down.
0x0008	Either ALT key down.
0x0010	SCROLL LOCK key turned on.
0x0020	NUMLOCK key turned on.
0x0040	CAPSLOCK key turned on.
0x0080	INSERT key turned on.
0x0100	Left CONTROL key down.
0x0200	Left ALT key down.
0x0400	Right CONTROL key down.
0x0800	Right ALT key down.
0x1000	SCROLL LOCK key down.
0x2000	NUMLOCK key down.
0x4000	CAPSLOCK key down.
0x8000	SYSREQ key down.

time Specifies the time stamp of the keystroke in milliseconds.

Comments

If the shift report input mode is enabled, the retrieved keystroke information may specify only a shift-state change and no character input.

See Also

READCHAR

■ **int DosDevIOCtl(0L, *pbCommand*, 0x0060, 0x000B, *hDevice*)**
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The QUERYMONSUPPORT function queries a device driver for monitor support.

Parameter	Description
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.

hDevice Identifies the device to receive the device-control function. The handle must have been previously created using the **DosOpen** function.

Return Value

The return value is zero if the device supports character monitors. Otherwise, it is the error value, “monitors not supported” (ERROR_MONITORS_NOT_SUPPORTED).

■ **int DosDevIOCtl**(*pkkiBuffer*, *pcRecords*, **0x0074**, **0x0004**, *hDevice*)
PKBDKEYINFO *pkkiBuffer*; pointer to buffer for keystrokes
PUSHORT *pcRecords*; pointer to variable for record count
HFILE *hDevice*; device handle

The READCHAR function retrieves one or more character data records from the keyboard input buffer for the screen group of the active process.

The 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 the *pcRecords* parameter is 0x0000, the function waits for the requested number of keystrokes, that is, it blocks the calling process until all records have been read. If the variable is 0x8000, 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 cooked; it sets the sign bit to 1 if the input mode is raw.

Parameter	Description
<i>pkkiBuffer</i>	Points to the buffer to receive the character data records. The buffer size must be at least as large as the size of an individual record multiplied by the requested number of records to be read. For a full description of a character data record, see the following “Structures” section.
<i>pcRecords</i>	Points to the variable containing the number of records to be read. When the function returns, it copies the actual number of records retrieved to the variable.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pkkiBuffer* parameter has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

Field	Description										
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: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>A shift key is received (valid only in raw mode when shift report is on).</td></tr><tr><td>0x0020</td><td>Conversion requested.</td></tr><tr><td>0x0040</td><td>Final character received.</td></tr><tr><td>0x0080</td><td>Interim character received.</td></tr></table>	Value	Meaning	0x0001	A shift key is received (valid only in raw mode when shift report is on).	0x0020	Conversion requested.	0x0040	Final character received.	0x0080	Interim character received.
Value	Meaning										
0x0001	A shift key is received (valid only in raw mode when shift report is on).										
0x0020	Conversion requested.										
0x0040	Final character received.										
0x0080	Interim character received.										
bNlsShift	Specifies a reserved value. It must be zero.										
fsState	Specifies the state of the shift keys. It can be any combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>SHIFT key up.</td></tr><tr><td>0x0001</td><td>Right SHIFT key down.</td></tr></table>	Value	Meaning	0x0000	SHIFT key up.	0x0001	Right SHIFT key down.				
Value	Meaning										
0x0000	SHIFT key up.										
0x0001	Right SHIFT key down.										

0x0002	Left SHIFT key down.
0x0004	Either CONTROL key down.
0x0008	Either ALT key down.
0x0010	SCROLL LOCK key turned on.
0x0020	NUMLOCK key turned on.
0x0040	CAPSLOCK key turned on.
0x0080	INSERT key turned on.
0x0100	Left CONTROL key down.
0x0200	Left ALT key down.
0x0400	Right CONTROL key down.
0x0800	Right ALT key down.
0x1000	SCROLL LOCK key down.
0x2000	NUMLOCK key down.
0x4000	CAPSLOCK key down.
0x8000	SYSREQ key down.

time Specifies the time stamp of the keystroke in milliseconds.

See Also

KBDCHARIN

■ **int DosDevIOCtl(*pmeiEvent*, *pfWait*, 0x0063, 0x0007, *hDevice*)**
PMOUEVENTINFO *pmeiEvent*; pointer to buffer for event information
PUSHORT *pfWait*; pointer to wait/no-wait flag
HFILE *hDevice*; device handle

The READEVENTQUE function reads the event queue for the pointing device.

Parameter	Description
<i>pmeiEvent</i>	Points to the structure to receive the event data. For a full description, see the following “Structures” section.
<i>pfWait</i>	Points to the variable specifying how to read from the queue if no event is available. If the variable is 0x0000, the function returns immediately without an event if no event is available. If the variable is 0x0001, the function waits until an event is available.

hDevice Identifies the mouse device to receive the device-control function. The handle must have been previously created using the **DosOpen** function.

Return Value

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

Structures

The structure pointed to by the *pmeiEvent* parameter has the following form:

```
typedef struct _MOUEVENTINFO {
    USHORT fs;
    ULONG Time;
    USHORT row;
    USHORT col;
} MOUEVENTINFO;
```

Field	Description																		
fs	Specifies the current event for the device. It can be a combination of the following values:																		
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>No buttons down.</td></tr><tr><td>0x0001</td><td>Mouse moved with no buttons down.</td></tr><tr><td>0x0002</td><td>Mouse moved with button 1 down.</td></tr><tr><td>0x0004</td><td>Button 1 down.</td></tr><tr><td>0x0008</td><td>Mouse moved with button 2 down.</td></tr><tr><td>0x0010</td><td>Button 2 down.</td></tr><tr><td>0x0020</td><td>Mouse moved with button 3 down.</td></tr><tr><td>0x0040</td><td>Button 3 down.</td></tr></table>	Value	Meaning	0x0000	No buttons down.	0x0001	Mouse moved with no buttons down.	0x0002	Mouse moved with button 1 down.	0x0004	Button 1 down.	0x0008	Mouse moved with button 2 down.	0x0010	Button 2 down.	0x0020	Mouse moved with button 3 down.	0x0040	Button 3 down.
Value	Meaning																		
0x0000	No buttons down.																		
0x0001	Mouse moved with no buttons down.																		
0x0002	Mouse moved with button 1 down.																		
0x0004	Button 1 down.																		
0x0008	Mouse moved with button 2 down.																		
0x0010	Button 2 down.																		
0x0020	Mouse moved with button 3 down.																		
0x0040	Button 3 down.																		
Time	Specifies the time the event was generated.																		
row	Specifies the <i>x</i> -coordinate of the pointer.																		
col	Specifies the <i>y</i> -coordinate of the pointer.																		

1

Field	Description
bCommand	Specifies the type of track layout. If it 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.
head	Specifies the physical head on the drive to perform the read operation.
cylinder	Specifies the cylinder number to perform the read operation.
firstSector	Specifies the logical sector number to start the read 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 to read, up to the maximum number specified in the track table. The READPHYSTRACK function does not step heads and tracks.
TrackTable[N]	Specifies the track-layout table. It is an array of structures that contain the numbers and sizes of the sectors in the track. The sectorNumber field specifies the section number. The sectorSize field specifies the size of the sector. The first element defines the sector number and size in bytes of the first sector on the track. The number of elements depends on the number of sectors on the track.

Comments

The device driver will not correctly read sectors of sizes other than 512 bytes if the read operation would generate a direct-memory-access (DMA) violation error. Programs must make sure that this error will not occur. The READPHYSTRACK function also affects the logical units on the physical disk drive.

See Also

WRITEPHYSTRACK, WRITETRACK

■ **int DosDevIOCtl(*pbBuffer*, *pbCommand*, 0x0064, 0x0008, *hDevice*)**
PBYTE *pbBuffer*; pointer to buffer for data
PBYTE *pbCommand*; pointer to buffer with command
HFILE *hDevice*; device handle

The READTRACK function reads from a track on a drive. The function performs the read operation on the device specified in the request. The

track table passed in the call determines the sector number, which is passed to the disk controller for the operation. When the sectors are odd numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is read.

Parameter	Description
<i>pbBuffer</i>	Points to the buffer to receive the data read from the track. The buffer must be large enough to hold the specified number of bytes.
<i>pbCommand</i>	Points to the structure containing the information about the read operation. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbCommand* parameter has the following form:

```
struct {
    BYTE    bCommand;
    USHORT  head;
    USHORT  cylinder;
    USHORT  firstSector;
    USHORT  cSectors;
    struct {
        USHORT  sectorNumber;
        USHORT  sectorSize;
    } TrackTable[N];
};
```

Field	Description
bCommand	Specifies the type of track layout. If it 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.
head	Specifies the physical head on the drive to perform the read operation.

cylinder	Specifies the cylinder number to perform the read operation.
firstSector	Specifies the logical sector number to start the read 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 to read, up to the maximum number specified in the track table. The READTRACK function does not step heads and tracks.
TrackTable[N]	Specifies the track-layout table. It is an array of structures that contains the numbers and sizes of the sectors on the track. The sectorNumber field specifies the sector number. The sectorSize field specifies the size of the sector. The first element defines the sector number and size in bytes of the first sector on the track. The number of elements depends on the number of sectors on the track.

Comments

The device driver will not correctly read sectors of sizes other than 512 bytes if the read operation would generate a direct-memory-access (DMA) violation error. Programs must make sure that this error will not occur.

See Also

WRITETRACK

- int** **DosDevIOCtl**(**0L**, *pbCommand*, **0x0002**, **0x0008**, *hDevice*)
PBYTE *pbCommand*; pointer to variable with command.
HFILE *hDevice*; device handle

The REDETERMINEMEDIA function redetermines the media on a block device and updates the volume in the drive. It is normally issued after the volume ID information on the volume has been changed (such as by formatting the disk). The function should be called only if the volume is locked.

Parameter	Description
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

■ **int DosDevIOCtl(*pusInfo*, *pbCommand*, 0x0040, 0x000A, *hDevice*)**
PUSHORT *pusInfo*; pointer to monitor-register information
PBYTE *pbCommand*; pointer to command
HFILE *hDevice*; device handle

The REGISTERMONITOR function registers a monitor.

Parameter	Description
<i>pusInfo</i>	Points to the structure containing the monitor-register information. For a full description, see the following "Structures" section.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pusInfo* parameter has the following form:

```
struct {
    USHORT position;
    USHORT index;
    ULONG  buffer;
    USHORT offset;
};
```

Field	Description
position	Specifies the position-flag parameter used in the DosMonReg function. The field can be one of the following values:

	Value	Meaning
	0x0000	No positional preference
	0x0001	Front of list
	0x0002	Back of list
index	Specifies a device-specific value.	
buffer	Points to the monitor input buffer that is initialized by the monitor dispatcher and used by the DosMonRead function.	
offset	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

- int DosDevIOCtl(0L, *pnprBuffer*, 0x0058, 0x0007, *hDevice*)**
PNOPTRRECT *pnprBuffer*; points to buffer with exclusive rectangle
HFILE *hDevice*; device handle

The REMOVEPTR function specifies the exclusion rectangle to be used by the device driver. The exclusion rectangle specifies an area on the screen in which the pointer-draw routine will not draw the pointer.

Parameter	Description
<i>pnprBuffer</i>	Points to the structure containing the dimensions of the exclusion rectangle. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pnprBuffer* parameter has the following form:

```
typedef struct _NOPTRRECT {
    USHORT row;
    USHORT col;
    USHORT cRow;
    USHORT cCol;
} NOPTRRECT;
```

Field	Description
row	Specifies the starting <i>x</i> -coordinate.
col	Specifies the starting <i>y</i> -coordinate.
cRow	Specifies the height of the exclusion rectangle.
cCol	Specifies the width of the exclusion rectangle.

These fields may be specified in either character or pixel values as long as the orientation is preserved across all values in the exclusion rectangle.

Comments

The pointer is not drawn in the exclusion rectangle until a different area is specified with another call of this function.

If the exclusion rectangle is defined as the entire screen, pointer-draw operations are effectively disabled for the entire screen group.

■ **int DosDevIOCtl(0L, *pbNotify*, 0x0052, 0x0007, *hDevice*)**
PBYTE *pbNotify*; pointer to buffer with screen group
HFILE *hDevice*; device handle

The SCREENSWITCH function notifies the mouse device driver that a screen group switch is about to take place. It also sets a system pointer-draw enable/disable flag, effectively locking out any pointer drawing until the flag is cleared by the ALLOWPTRDRAW function (0x0007, 0x0050).

Parameter	Description
<i>pbNotify</i>	Points to the structure containing the notification type and screen-group number. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbNotify* parameter has the following form:

```
struct {  
    USHORT sgNumber;  
    USHORT fTerminate;  
};
```

Field	Description
sgNumber	Specifies the screen-group number for notification action. The number 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 the notification type of the switch. If it is 0xFFFF, the given screen group is terminating. If it is greater than 0x0000, the given screen group is receiving control.

- **USHORT DosDevIOctl(0L, *pusBitRate*, 0x0041, 0x0001, *hDevice*)**
PUSHORT *pusBitRate*; pointer to variable with baud rate
HFILE *hDevice*; device handle

The SETBAUDRATE function sets the baud (bit) rate for the specified serial device.

Parameter	Description
<i>pusBitRate</i>	Points to the variable that contains the baud rate. It can be any one of the following values: 110, 150, 300, 600, 1200, 2400, 4800, 9600, or 19200.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. The function returns the “general failure” error code (ERROR_GEN_FAILURE) if the specified baud rate is out of range.

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

GETBAUDRATE

-
- USHORT DosDevIOCtl(*pfCommErr*, 0L, 0x0045, 0x0001, *hDevice*)
PUSHORT *pfCommErr*;
HFILE *hDevice*;

pointer to variable for error value
device handle

The 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 will then resume transmitting characters, taking into account all the other reasons why it may or may not transmit characters.

Parameter	Description										
<i>pfCommErr</i>	Points to the variable that receives the communication status of the device. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.</td></tr><tr><td>0x0002</td><td>Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.</td></tr><tr><td>0x0004</td><td>The hardware detected a parity error.</td></tr><tr><td>0x0008</td><td>The hardware detected a framing error.</td></tr></table> The function sets the variable to zero if it encounters an error.	Value	Meaning	0x0001	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.	0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.	0x0004	The hardware detected a parity error.	0x0008	The hardware detected a framing error.
Value	Meaning										
0x0001	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.										
0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.										
0x0004	The hardware detected a parity error.										
0x0008	The hardware detected a framing error.										

SETBREAKOFF

hDevice Identifies the serial device that receives the device-control function. The handle must have been previously created by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

See Also

SETBREAKON

- **USHORT** **DosDevIOCtrl**(*pfCommErr*, 0L, 0x004B, 0x0001, *hDevice*)
PUSHORT *pfCommErr*; pointer to variable for error value
HFILE *hDevice*; device handle

The 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 will 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.

Parameter	Description
<i>pfCommErr</i>	Points to the variable that receives the communication status of the device. It can be a combination of the following values:
Value	Meaning
0x0001	Receive queue overrun. There is no room in the device-driver receive queue to put a character read in from the receive hardware.
0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.
0x0004	The hardware detected a parity error.
0x0008	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 previously created by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

Comments

Closing the device turns off the break character if there are no outstanding open device handles.

See Also

SETBREAKOFF

■ **USHORT DosDevIOctl(0L, *pusDCB*, 0x0053, 0x0001, *hDevice*)**
PUSHORT *pusDCB*; pointer to buffer with device-control information
HFILE *hDevice*; device handle

The SETDCBINFO function sets device-control block information.

Parameter	Description
<i>pusDCB</i>	Points to the structure that receives the device-control block information. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. When an error occurs, the function returns a “general failure” error code (ERROR_GEN_FAILURE) and the device-control block characteristics of the device driver for this serial device remain unchanged.

Structures

The structure pointed to by the *pusDCB* parameter has the following form:

```
struct {
    USHORT usWriteTimeout;
    USHORT usReadTimeout;
    BYTE   bFlags1
    BYTE   bFlags2
    BYTE   bFlags3
    BYTE   bErrorReplacementChar;
    BYTE   bBreakReplacementChar;
    BYTE   bXONChar;
    BYTE   bXOFFChar;
};
```

Field	Description
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-hundredth 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.
bFlags1	Specifies the control and handshaking modes for the device. It can be a combination of the following values:
Value	Meaning
0x01	Enable the data-terminal-ready (DTR) control mode. For a full description, see Comment 1 in the following "Comments" section.
0x02	Enable the data-terminal-ready (DTR) input handshaking mode. For a full description, see Comment 1 in the following "Comments" section.
0x08	Enable output handshaking using the clear-to-send (CTS) signal. For a full description, see Comment 3 in the following "Comments" section.
0x10	Enable output handshaking using the data-set-ready (DSR) signal. For a full description, see Comment 3 in the following "Comments" section.

0x20 Enable output handshaking using the data-carrier-detect (DCD) signal. For a full description, see Comment 3 in the following “Comments” section.

0x40 Enable input sensitivity using the data-set-ready (DSR) signal. For a full description, see Comment 4, in the following “Comments” section.

The values 0x03, 0x04, and 0x80 are reserved.

bFlags2

Specifies flow control and replacement character modes of the device. It can be a combination of the following values:

Value	Meaning
0x01	Enable automatic transmit flow control (XON/XOFF). For a full description, see Comment 2 in the following “Comments” section.
0x02	Enable automatic receive flow control (XON/XOFF). For a full description, see Comment 2 in the following “Comments” section.
0x04	Enable error replacement character. For a full description, see Comment 6 in the following “Comments” section.
0x08	Enable null stripping (remove null bytes). For a full description, see Comment 7 in the following “Comments” section.
0x10	Enable break replacement character. For a full description, see Comment 8 in the following “Comments” section.
0x40	Enable the request-to-send (RTS) control mode. For a full description, see Comment 1 in the following “Comments” section.
0x80	Enable the request-to-send (RTS) input handshaking mode. For a full description, see Comment 1 in the following “Comments” section.
0xC0	Enable toggling on transmit mode. For a full description, see Comment 1 in the following “Comments” section.

If a specific value is not used, the corresponding feature is disabled. The value 0x20 is reserved.

bFlags3 Specifies the time-out processing for the device. It can be a combination of the following values:

Value	Meaning
0x01	Enable write infinite time-out processing. For a full description, see Comment 9 in the following “Comments” section.
0x02	Enable normal read time-out processing. For a full description, see Comment 10 in the following “Comments” section.
0x04	Enable wait-for-something read time-out processing. For a full description, see Comment 10 in the following “Comments” section.
0x06	Enable no-wait read time-out processing. For a full description, see Comment 10 in the following “Comments” section.

All other values are reserved. Exactly one read time-out value (0x02, 0x04, or 0x06) is required. The function returns an error value if a value is not given.

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.

Any values not specified for the **bFlags1**, **bFlags2**, and **bFlags3** fields are considered to be reserved and must not be used. Any attempt to use these values will result in a “general failure” error.

A program can prevent making unwanted changes to device modes by calling the 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 SETDCBINFO function.

Comments

Comment 1: Control of DTR and RTS

The device driver lets a program automatically control the setting of data-terminal-ready (DTR) and request-to-send (RTS) control modes in the following ways:

- Set RTS control mode to Toggling on Transmit.
- Set DTR and/or RTS control mode to Input Handshaking
- Set DTR and/or RTS control mode to Enable or Disable

A program can also request manual control over these modem-control signals.

Set RTS Control Mode to Toggling on Transmit

When bits 6 and 7 are set in the **Flags2** field, the device driver is in this automatic control mode of RTS. When the device driver is initialized, the RTS control mode is set to Enable, so initially the device driver is not in this automatic control mode of RTS.

This mode of operation of the device driver should only be enabled when the system is attached to devices which will not present data to the system receive hardware when RTS is on.

In this mode, the device driver will carry out the following tasks:

1. Always turn on RTS if a break is being transmitted.
2. Once data is in the transmit hardware buffer, the device driver will not turn RTS off until the transmit hardware has emptied its buffers.
3. Turn on RTS (if not already on) if there is data in the device-driver transmit queue or if there is an outstanding write request packet and one of the following conditions is true:
 - The device driver is allowed to transmit even if automatic transmit/receive flow control (XON/XOFF) is enabled. RTS must be turned on momentarily to transmit a character immediately if it is not normally allowed to transmit due to automatic transmit/receive flow control (XON/XOFF).
 - The device driver is allowed to transmit because it was not told to behave as if an XOFF character had been received using the STOP-TRANSMIT function (0x0001, 0x0047). The device driver will still need to turn on RTS momentarily to transmit a character immediately if not normally transmitting due to XOFF flow control considerations. The device driver will still need to turn on RTS momentarily to transmit an XON or XOFF due to automatic receive flow control if not normally transmitting due to XOFF flow control considerations.
4. Turn off RTS (if it is not already off) if either of the following conditions is true:

- There is no more data in the device-driver transmit queue (and no more data in write requests in progress), no queued write requests, and the transmit hardware has physically transmitted (at the physical RS232 interface) all the data that it has been given.
- The device driver is not allowed to transmit due to transmit/receive flow control (XON/XOFF) being enabled or due to being asked to behave as if an XOFF character had been received using the STOPTRANSMIT function (0x0001,0x0047). The device driver still needs to turn on RTS to transmit a character immediately, or XON/XOFF due to automatic receive flow control (XON/XOFF). RTS is never turned off until the transmit hardware has physically transmitted (at the physical RS232 interface) all the data that it has been given.

5. When this function is enabled, the device driver will control RTS appropriately as described above.

6. If this function is disabled (by choosing a new RTS control mode), then RTS will be controlled appropriately by the new RTS control mode that is inherently chosen when this RTS control mode is disabled.

7. The device driver will not examine any other modem-control signals before it turns RTS off or on.

An open request packet will not cause the device driver to change the RTS control mode that the device driver is in. The device driver will maintain the state of this mode of operation across open request packets.

When the device driver is in the RTS control mode “toggling on transmit,” the device driver will not allow the application to control RTS through the SETMODEMCTRL function (0x0001,0x0046).

Set DTR and/or RTS Control Mode to Input Handshaking

When bit 0 is clear and bit 1 is set in the **Flags1** field, the DTR control mode is set to Input Handshaking. When bit 6 is clear and bit 7 is set in the **Flags2** field, the RTS control mode is set to Input Handshaking.

When the device driver is initialized, the RTS and DTR control mode is set to Enable; so initially the device driver is not in this automatic control mode of RTS and DTR.

This mode of operation of the device driver should be set only when there is the possibility of a device-driver receive queue overrun and the system is attached to data terminal equipment that will stop transmitting data when the appropriate modem-control signals are turned off due to the cabling and the data terminal equipment characteristics.

Keep in mind that this mode can be set for either RTS or DTR, or both; the DTR and RTS control modes are processed independently.

In this mode, the device driver will carry out the following tasks:

1. Turn the appropriate modem-control signal(s) on when the device-driver receive queue is less than about half full.
2. Turn the appropriate modem-control signal(s) off when the device-driver receive queue gets close to full.
3. When this mode is first set, the device driver does not monitor the value of the appropriate modem-control signal(s) (DTR or RTS) when the queue size is between approximately half full and almost full.
4. When this function is enabled, the device driver determines the correct value of the modem-control signal(s) and controls them accordingly.
5. If this function is disabled (by choosing a new RTS and/or DTR control mode), RTS and/or DTR are controlled appropriately by the new RTS and/or DTR control mode that is inherently chosen when this mode is disabled.
6. The device driver does not examine any other modem-control signals before controlling DTR or RTS due to this mode.

An open request packet does not cause the device driver to change the RTS and DTR control modes that the device driver is in. The device driver maintains the state of these modes of operation across open request packets.

When the device driver is in the RTS control mode "Input Handshaking," the device driver does not allow the application to control RTS through the SETMODEMCTRL function (0x0001, 0x0046). When the device driver is in the DTR control mode "Input Handshaking," the device driver does not allow the application to control DTR through the SETMODEMCTRL function (0x0001, 0x0046).

Set DTR and/or RTS control mode to Enable or Disable

OPEN processing. When bits 0 and 1 of the **Flags1** field are both zero, the DTR control mode is set to Disable. When bit 0 is set and bit 1 is clear, the DTR control mode is set to Enable. When bits 6 and 7 are clear in the **Flags2** field, the RTS control mode is set to Disable. When bit 6 is set but bit 7 is clear in the **Flags2** field, the RTS control mode is set to Enable. When the device driver is initialized, the RTS and DTR control modes are both set to Enable; the value of the modem-control signals is off until the port gets an open request packet.

An open request packet does not cause the device driver to change the RTS and DTR control modes that the device driver is in. The device driver maintains the state of these modes of operation across open request packets.

Keep in mind that these modes can be set for either RTS or DTR, or both; the DTR and RTS control modes are processed independently. The following discussion covers what happens to RTS. The same discussion applies to DTR as well if the DTR control mode is set as described in the RTS discussion.

If the RTS control mode is set to Disable, then when the device driver receives an open request packet and the device is not already open (from a previous open without a close) then the RTS modem-control signal will be kept (turned) off during the open processing. If the RTS control mode is set to Enable, when the device driver receives an open request packet and the device is not already open (from a previous open without a close), the RTS modem-control signal will be turned on during open processing.

If the RTS control mode is set to Disable and the previous mode was not set to Disable, the RTS modem-control signal is turned off. If the RTS control mode is set to Disable and the previous mode was also set to Disable, this function has no effect on the RTS modem-control signal.

If the RTS control mode is set to Enable and the previous mode was not set to Enable, the RTS modem-control signal is turned on. If the RTS control mode is set to Enable and the previous mode was also set to Enable, this function has no effect on the RTS modem-control signal.

Tables 4.1 and 4.2 summarize this discussion:

Table 4.1

DTR or RTS: Input Handshaking

From Control Mode	To Control Mode	Effect on Modem Control
Disable	Disable	None
Disable	Enable	Turn on
Disable	Input handshaking	Auto, see input handshaking
Enable	Disable	Turn off
Enable	Enable	None
Enable	Input handshaking	Auto, see input handshaking
Input handshaking	Disable	Turn off
Input handshaking	Enable	Turn on
Input handshaking	Input handshaking	Auto, see input handshaking

Table 4.2

RTS Only: Toggle on Transmit (Toggle)

From Control Mode	To Control Mode	Effect on Modem Control
Disable	Toggle	Auto, see toggling on transmit
Enable	Toggle	Auto, see toggling on transmit
Input handshaking	Toggle	Auto, see toggling on transmit
Toggle	Disable	Turn off
Toggle	Enable	Turn on
Toggle	Input handshaking	Auto, see input handshaking
Toggle	Toggle	Auto, see toggling on transmit

Because the initial control mode of the device driver is Enable for RTS and DTR both modem-control signals will be turned on when the port is first opened.

If the device driver receives an open request packet and the device is already open, the device driver does not alter the value of the RTS and DTR modem-control signals regardless of the control mode.

Program Control of DTR and RTS. The application can explicitly turn DTR or RTS on or off (independently) with the SETMODEMCTRL function (0x0001,0x0046).

If the control mode of RTS is not set to Enable or Disable, the application may not control RTS with the SETMODEMCTRL function (0x0001, 0x0046) because the device driver is controlling the signal automatically (toggling on transmit or input handshaking). If the control mode of DTR is not set to Enable or Disable, the application may not control DTR with the SETMODEMCTRL function (0x0001,0x0046) because the device driver is controlling the signal automatically (input handshaking).

Close Processing. If the device driver receives a close request packet and the serial device is still open (from another open without a close), the device driver will not change the values of DTR or RTS.

If the device driver receives a close request packet (after processing a close request the port closes, from another open without a close), at the end of the close processing RTS and DTR are turned off by the device driver, after waiting the appropriate amount of time. For more information, see the SETMODEMCTRL function (0x0001,0x0046).

Comment 2: Automatic Flow Control: XON/XOFF Characters

When bit 0 is set in the **Flags2** field, the device driver is enabled for automatic transmit flow control. If bit 1 is set, the device driver is enabled for automatic receive flow control.

When the device driver is initialized, these bits are reset. Initially, the device driver is not enabled for automatic transmit or receive flow control.

An open request packet will not cause the device driver to change the enabling or disabling state of automatic transmit/receive flow control.

An open request packet, when the serial device is not already open (from a previous open without a close), causes the device driver to believe it has not received an XOFF character if automatic transmit flow control is enabled. The device driver assumes it has not transmitted an XOFF character if automatic receive flow control is enabled.

The discussion that follows states that the device driver will transmit XON or XOFF characters. There are reasons why the device driver may not be able to transmit an XON or XOFF (transmitting break, invalid output handshaking on modem control signals). It is also stated that the device driver will resume transmitting data. There are other potential reasons (not related to automatic transmit/receive flow control) why that may not be possible. See the GETCOMMSTATUS function (0x0001, 0x0064).

Automatic Transmit Flow Control (XON/XOFF)

When XON and XOFF flow control during transmission is enabled, the device driver stops sending data to the transmit hardware when an XOFF character is received, and resume sending data to the transmit hardware when an XON character is received. (The device still may not be able to transmit due to other reasons.) The device driver will still transmit characters due to the TRANSMITIMM function (0x0001, 0x0044). The device driver will still transmit XON and XOFF characters due to automatic receive flow control.

When the device driver is in this mode, it will not pass received XON and XOFF characters to the application. Instead, the device driver acts upon receiving these characters and throws them away.

The device driver may transmit additional characters before it recognizes an XOFF character which may be in the receive buffer of the hardware but which it has not read. If the system is relatively slow in responding to interrupts compared to the current baud rate, receive buffer overruns may not be occurring but the device driver may be slow in responding to an XOFF character.

If automatic transmit flow control is disabled (after being currently enabled) and transmission is not occurring due to an XOFF character being received or the STOPTRANSMIT function (0x0001, 0x0047) being requested, transmission will be resumed (although transmission may not resume for other reasons).

It is the program's responsibility to not fully close the port in a way that causes the device driver to illegally transmit characters when the port is reopened after being fully closed (first level open).

Output handshaking using modem-control signals is another way that the device driver can be told to stop transmitting. For a full description, see Comment 3.

This section described algorithms based on current hardware. The buffering characteristics discussed here are subject to change based upon different hardware support.

Automatic Receive Flow Control (XON/XOFF)

When XON and XOFF flow control during receive is enabled, the device driver transmits an XOFF character when its receive queue gets close to full, and an XON character when its receive queue is about half full. After the XOFF is sent, the serial device driver sends no characters until it sends an XON character due to automatic receive flow control. This is done to accommodate systems that interpret the first character received after an XOFF character as an XON character, regardless of the actual character. The device driver still transmits characters due to the TRANSMITIMM function (0x0001,0x0044).

"Close to full" and "about half full" are very approximate statements. The low-level design determines the exact conditions, which may change from release to release of the device driver.

The device driver is not able to transmit XOFF or XON characters if it is transmitting a break, and the device driver is not able to automatically transmit XOFF or XON characters if it is enabled for output handshaking with certain modem-control signals and those modem-control signals are not on. This could cause a deadlock if the device driver tries to transmit an XON character when it cannot. The device driver remembers that it wanted to transmit an XOFF or XON character and will still do so when transmit conditions permit, assuming the receive queue conditions still warrant it.

The device driver does not monitor characters being transmitted by write request packets to see if any of them are XON or XOFF characters. The device driver also does not monitor characters transmitted immediately using the TRANSMITIMM function (0x0001,0x0044). For example, the device driver does not stop transmitting characters if the application causes the device driver to explicitly transmit an XOFF character.

If automatic receive flow control is enabled (after being disabled), the device driver immediately checks the receive queue level to see if an XOFF character needs to be transmitted. An XON character is never transmitted immediately due to this function being enabled. The device driver will only automatically transmit an XON character after it has automatically transmitted an XOFF character.

If automatic receive flow control is disabled (after being enabled) and transmission is not occurring due to an XOFF character being automatically transmitted, the device driver transmits an XON character and transmission will be resumed if possible (although, transmission may not be taking place for other reasons).

If the device driver previously automatically transmitted an XOFF character and a close request packet is received (after processing this close request the port closes, from another open without a close), the device driver automatically transmits an XON character if possible.

It is the application's responsibility to not close the port in a way that causes the device driver to illegally transmit characters or the communications link to be in a deadlock state when the port is reopened after being fully closed.

Input handshaking using modem-control signals is another way that the device driver can tell another device to stop transmitting. For a full description, see Comment 1.

XON and XOFF Characters

The value of these bytes in the device-control block determines the value of the XON and XOFF characters used for automatic transmit and receive flow control.

When the device driver is first initialized the XON character is 0x11 and the XOFF character is 0x13. An open request packet (when the serial device is not already open, from a previous open without a close), causes the XON character to be set to 0x11 and the XOFF character to be set to 0x13.

If the XON and XOFF characters are set equal with this function, the results are undefined.

Comment 3: Output Handshaking Using CTS, DSR, DCD

Bits 3, 4, and 5 of the **Flags1** field control output handshaking using CTS, DSR, and DCD, respectively. If the bit is set, output handshaking for the appropriate modem-control signal is enabled.

Keep in mind that this mode can be enabled for any combination of CTS, DSR, or DCD because bits 3, 4, and 5 of the **Flags1** field are processed independently.

When the device driver is initialized, bits 3 and 4 are set and bit 5 is reset (of the **Flags1** field); initially, the device driver is enabled for output handshaking using CTS and DSR but disabled for output handshaking using DCD.

Except for attachment to special devices and/or special cables, output handshaking using DCD should never be enabled.

Disabling output handshaking using CTS and/or DSR causes unexpected results when the system is attached to data terminal devices or data communications devices that toggle CTS and/or DSR in order to control the ability of the system to transmit data.

If the device driver is enabled for this mode of operation, the device driver is affected in the following manner if the appropriate modem signal(s) are off:

1. The device driver is unable to move data from the device driver transmit queue to the transmit hardware.
2. The device driver is unable to transmit a character (TRANSMITMM (0x0001,0x0044)) so the character is remembered by the device driver.
3. The device driver is unable to automatically transmit XON and XOFF characters. The device driver may wish to transmit XON and XOFF characters as a result of automatic receive flow control being enabled.
4. The device driver still generates a break immediately, if requested.
5. The value of CTS, DSR, and DCD do not affect how the device driver controls RTS and DTR.

An open request packet does not cause the device driver to change the value of bits 3, 4, and 5 of the **Flags1** field. The device driver maintains the state of this mode of operation across open request packets.

On devices with a transmit holding register and a transmit shift register, the transmit holding register will always be given another character to transmit when it empties (even though a character may still be in the transmit shift register) unless the device driver determines that is not allowed to transmit any more.

The device driver will always attempt to detect a change in the modem status signals (CTS, DSR, DCD) before transmitting more data. This feature prevents a temporary elongation of interrupt latency from not allowing the device driver to recognize a change in a modem-control

signal. (The modem-control signal change happened many character times before the transmit hardware is requesting that another character be given to it.)

Comment 4: Input Sensitivity Using DSR

Bit 6 of the **Flags1** field controls input sensitivity using DSR. If the bit is set, input sensitivity using DSR is enabled.

When the device driver is initialized, bit 6 of the **Flags1** field is set; initially, the device driver is enabled for input sensitivity using DSR.

Disabling input sensitivity using DSR causes unexpected results when the system is attached to data terminal devices or data communications devices that toggle DSR when they generate spurious data that they do not wish the system to receive.

If the device driver is enabled for this mode of operation, the device driver throws away all data input from the receive hardware if DSR is off.

If the device driver processes a change in the DSR modem-control signal from on to off, or from off to on, at the same time that it inputs a character from the receive hardware, the device driver still accepts that last character(s). This prevents a temporary elongation of interrupt latency from causing the device driver to discard a valid character(s). However, this could cause the device driver to attempt to process invalid data for one service period of the receive hardware. This requires that the change in the modem-control signal be processed before the device driver attempts to receive data from the receive hardware (for a full description, see Comment 3), or the received data is saved until a change in modem status (during the same hardware service instance) can be determined.

An open request packet does not cause the device driver to change the value of bit 6 of the **Flags1** field. The device driver maintains the state of this mode of operation across open request packets.

Comment 6: Error Replacement Character

Bit 2 in the **Flags2** field controls the enabling of error replacement character processing. If the bit is set, the error replacement character processing is enabled.

When the device driver is initialized this bit is reset, initially, the device driver is not enabled for the error replacement character. An open request packet (when the serial device is not already open, from a previous open without a close, first level open) causes this bit to be reset, disabling error replacement character processing.

When the device driver is initialized, the error replacement character is 0x00. An open request packet (when the serial device is not already open, from a previous open without a close) causes the error replacement character to be set back to 0x00.

If error replacement character processing is disabled, the following occurs:

1. If a parity or framing error occurs, the character that had the error is available in the receive hardware buffer; it is placed in the device driver receive queue.
2. If a hardware or receive queue overrun occurs, nothing is placed in the receive queue to designate an overrun.

If error replacement character processing is enabled, the following occurs:

1. If a parity or framing error occurs, the error replacement character is placed in the device-driver receive queue (instead of the character in the receive hardware buffer if it was available).
2. If a hardware buffer overrun occurs, the error replacement character is placed in the device-driver receive queue to mark the position that a receive overrun occurred. If valid data is in the receive hardware buffer, it is still placed in the device-driver receive queue. The processing of the valid data takes place after the hardware buffer overrun condition is recorded in the device-driver receive queue.
3. If a device-driver receive queue overrun occurs, the last character in the receive queue is replaced with the error replacement character. This allows the application to know the position where the error occurred. This error replacement (if enabled) always takes precedence over an error replacement or break replacement event that occurred at the same character time.

Regardless of whether error replacement character processing is enabled, null stripping and checking for XON/XOFF characters do not occur if the character had an error.

This function can be used to change the error replacement character by changing the byte representing the error replacement character.

Comment 7: Null Stripping

Bit 3 in the **Flags2** field controls the enabling of null-stripping processing. If this bit is set, null-stripping processing is enabled.

When the device driver is initialized, this bit is reset. Initially, the device driver is not enabled for null stripping. An open request packet (when the serial device is not already open, from a previous open without a close, first level open) causes this bit to be reset, disabling null stripping.

If the device driver is enabled for null stripping, when characters are read in from the receive hardware any characters (non-error or non-break) with a value of 0x00 are thrown away, are not checked even if the XON or XOFF character has been set to 0x00, and are not placed in the device-driver receive queue.

Simultaneously setting the XON or XOFF character to 0x00, enabling automatic transmit flow control, and enabling null stripping may cause unexpected results but is not considered an error condition by the device-driver error checking logic.

Comment 8: Break Replacement Character

Bit 4 in the **Flags2** field controls the enabling of break replacement character processing. If the bit is set, the break replacement character processing is enabled.

When the device driver is initialized, this bit is reset. Initially, the device driver is not enabled for the break replacement character. An open request packet (when the serial device is not already open, from a previous open without a close) causes this bit to be reset, disabling break replacement character processing.

When the device driver is initialized, the break replacement character is 0x00. An open request packet (when the serial device is not already open, from a previous open without a close, first level open) causes the break replacement character to be set back to 0x00.

If break replacement character processing is disabled, the device driver does not place any character in the device driver receive queue when it detects a break condition on the line. A detected break condition has no effect on XON/XOFF character detection.

If break replacement character processing is enabled, when the device driver detects a break condition it places the break replacement character in the device-driver receive queue.

If break replacement character processing is enabled, null stripping and checking for XON/XOFF characters does not operate on the break replacement character.

This function can be used to change the break replacement character by changing the byte representing the break replacement character.

If a parity or framing error is generated due to the reception of a break, error replacement processing is not done (except for the overrun condition); break replacement processing is done.

Comment 9: Write Timeout

Bit 0 in the **Flags3** field controls the characteristics of write time-out processing. If the bit is 0, write time-out processing uses the value in the **usWriteTimeout** field in the device-control block. If the bit is 1, write time-out processing is infinite.

The value in the **usWriteTimeout** field is in 0.01 second units (based on 0, where 0 = 0.01 seconds). The device driver is considered doing normal write time-out processing when the field is used for write time-out processing.

During normal write time-out processing, if the device driver does not give any data to the transmit hardware (from the transmit queue) within the period of time specified by the **usWriteTimeout** field (due to some reason that prevents the device driver from transmitting data), the request is completed. The accuracy of the time-out period may be determined by the request packet being blocked in the device driver and how long it takes for the thread to be dispatched once it is made ready by the time-out period expiring; or the accuracy of the time-out period may be determined by the accuracy of the device-driver timer tick processing. If any data are given to the transmit hardware in that time-out period, the specified period of time will be waited on again, to see if any more data are transmitted.

If the time-out period is changed by this function (or to infinite time-out), the new time may take effect immediately or may take effect after the next character is written.

During write infinite time-out processing, the request is not completed until all the data from the request has been given to the transmit hardware. The thread of the write request will not return to the system until the request completes. The device driver checks to see if a function has changed the write time-out processing characteristics at least every minute but could be almost immediately (accuracy may be determined by the request packet being blocked and/or by device-driver timer ticks). This ensures that the device driver periodically checks to see if write infinite time-out processing has been changed to normal write time-out processing.

As discussed previously, the write time-out characteristics can be changed in the middle of the processing of a write request and the new time-out attributes are guaranteed to eventually take effect. When the device driver initializes, normal write time-out processing is in effect.

When the device driver receives an open request packet for the port and the port is not already open (from a previous open without a matching close), the value in the write time-out word is set to 1 minute but the current write time-out processing characteristics (normal or infinite) are not affected.

Comment 10: Read Timeout

Bits 1 and 2 in the **Flags3** field control the read time-out processing characteristics of the device driver. The three possible types of read time-out processing are listed below:

- Normal (Bits 2,1= 0,1)
- Wait For Something (Bits 2,1= 1,0)
- No Wait (Bits 2,1= 1,1)

The value in the **usReadTimeout** field is in 0.01 second units (based on 0, where 0 = 0.01 seconds). The device driver uses the value in the field for normal- and wait-for-something read time-out processing. The accuracy of the time interval may be determined by the request being blocked in the device driver and/or by device-driver timer ticks.

If the device driver is doing normal-read time-out processing, the device driver waits as long as the value in the **usReadTimeout** field. The request is completed after that interval of time elapses, if no more data has been received for the request. If any data is received by the device driver (from the receive hardware) for the request (including XON/XOFF characters), the specified period of time will be waited on again (for more data to arrive).

If the device driver is doing no-wait read time-out processing, the device driver does not wait for any data to be available in the receive queue. When the device driver tries to move data from the receive queue to the request, the request will complete. Whatever data is available in the receive queue at that time is the amount of data that will be moved to the request.

If the device driver is doing wait-for-something read time-out processing, the device driver processes the request initially as if it had no-wait time-out processing. If no data is available at the time the request is completed, due to no-wait processing, the request is not completed. Instead, the request waits for data to be available before completing the request. However, the device driver does enter normal-read time-out processing for this request, so if no data is available after the normal time-out processing interval, the request is completed anyway. The request never waits longer than it would for normal-read time-out processing.

The read time-out processing characteristics that apply to a given read request are not determined until the device driver begins processing that request. Once the device driver begins processing that request, a change to the read time-out processing characteristics of the device driver between wait-for-something and normal-read time-out processing may or may not take effect for the current read request being processed. If the time-out

period is changed by this function, the new time-out period may take effect immediately, or it may take effect after the next character is received from the receive hardware.

When the device driver initializes, normal read time-out processing is in effect.

When the device driver receives an open request packet for the port and the port is not already open (from a previous open without a matching close), the value in the write time-out word is set to 1 minute and normal read time-out processing characteristics are put into effect.

See Also

GETDCBINFO

```
■ int DosDevIOCtl(pbBPB, pbCommand, 0x0043, 0x0008, hDevice)
  PBYTE pbBPB;                pointer to buffer with BPB
  PBYTE pbCommand;            pointer to buffer with command
  HFILE hDevice;              device handle
```

The SETDEVICEPARAMS function sets the device parameters for MS OS/2 block devices. The device driver maintains two BIOS parameter blocks (BPB) for each drive. One block is the current BPB that corresponds to the media. The other block is a recommended BPB, based on the type of media that corresponds to the physical device. For example, a high-density disk drive has a BPB for a 96-tpi (tracks per inch) floppy disk; a low-density disk drive has a BPB for a 48-tpi floppy disk.

Parameter	Description						
<i>pbBPB</i>	Points to the structure containing the device parameters to be set for the drive. For a full description, see the following "Structures" section.						
<i>pbCommand</i>	Points to the variable containing the command description. It can be one of the following values:						
<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x00</td><td>Builds the BIOS parameter block (BPB) from the medium for all subsequent build BPB requests.</td></tr><tr><td>0x01</td><td>Changes the default BPB for the physical device.</td></tr></table>		Value	Meaning	0x00	Builds the BIOS parameter block (BPB) from the medium for all subsequent build BPB requests.	0x01	Changes the default BPB for the physical device.
Value	Meaning						
0x00	Builds the BIOS parameter block (BPB) from the medium for all subsequent build BPB requests.						
0x01	Changes the default BPB for the physical device.						

0x02 Changes the BPB for the medium to the specified BPB. It also returns the new BPB as the BPB for the medium for all subsequent build BPB requests.

hDevice Identifies the device that receives the device-control function. The handle must have been previously created by using the **DosOpen** function.

Return Value

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

Structures

The structure pointed to by the *pbBPB* parameter has the following form:

```
struct {
    USHORT usBytesPerSector;
    BYTE   bSectorsPerByte;
    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;
};
```

Field	Description
usBytesPerSector	Specifies the bytes per sector.
bSectorsPerByte	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.
- cCylinders** Specifies the number of cylinders defined for the physical device.
- bDeviceType** Specifies the physical layout of the specified device. It can be one of the following values:

Value	Meaning
0x0000	48 tracks per inch, low-density floppy disk drive
0x0001	96 tracks per inch, high-density floppy disk drive
0x0002	3.5-inch (720K) floppy disk drive
0x0003	8-inch, single-density floppy disk drive
0x0004	8-inch, double-density floppy disk drive
0x0005	Hard disk
0x0006	Tape drive
0x0007	Other (unknown type of device)

- fDeviceAttr** Specifies information about the drive. It can be a combination of the following values:

Value	Meaning
0x0001	Removable media.
0x0002	Media detects changes.

See Also

GETDEVICEPARAMS

SETEVENTMASK

- | | |
|---|-------------------------------|
| int DosDevIOCtl(0L, <i>pfEvent</i> , 0x0054, 0x0007, <i>hDevice</i>) | |
| PUSHORT <i>pfEvent</i> ; | pointer to variable for event |
| HFILE <i>hDevice</i> ; | device handle |

The SETEVENTMASK function sets the pointing-device event mask.

Parameter	Description																
<i>pfEvent</i>	Points to the variable containing the event mask. It can be a combination of the following values:																
	<table border="1"> <thead> <tr> <th>Value</th><th>Meaning</th></tr> </thead> <tbody> <tr> <td>0x0001</td><td>All mouse motion, no buttons are pressed.</td></tr> <tr> <td>0x0002</td><td>Motion with button 1 pressed.</td></tr> <tr> <td>0x0004</td><td>Button 1 pressed.</td></tr> <tr> <td>0x0008</td><td>Motion with button 2 pressed.</td></tr> <tr> <td>0x0010</td><td>Button 2 pressed.</td></tr> <tr> <td>0x0020</td><td>Motion with button 3 pressed.</td></tr> <tr> <td>0x0040</td><td>Button 3 pressed.</td></tr> </tbody> </table>	Value	Meaning	0x0001	All mouse motion, no buttons are pressed.	0x0002	Motion with button 1 pressed.	0x0004	Button 1 pressed.	0x0008	Motion with button 2 pressed.	0x0010	Button 2 pressed.	0x0020	Motion with button 3 pressed.	0x0040	Button 3 pressed.
Value	Meaning																
0x0001	All mouse motion, no buttons are pressed.																
0x0002	Motion with button 1 pressed.																
0x0004	Button 1 pressed.																
0x0008	Motion with button 2 pressed.																
0x0010	Button 2 pressed.																
0x0020	Motion with button 3 pressed.																
0x0040	Button 3 pressed.																
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.																

Return Value

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

- | | | |
|----------------|---|-------------------------------------|
| int | DosDevIOCtl(0L, <i>pusScreenGrp</i>, 0x0055, 0x0004, <i>hDevice</i>) | |
| PUSHORT | <i>pusScreenGrp</i> ; | pointer to buffer with screen group |
| HFILE | <i>hDevice</i> ; | device handle |

The SETFGNDSCREENGRP function sets the new foreground screen group. When the keyboard switches to the new screen group, it switches the keyboard to the shift state, input buffer, and monitor chain defined for the specified screen group.

This function is reserved for the session manager.

Parameter	Description
<i>pusScreenGrp</i>	Points to the structure containing the screen-group identifier of the new foreground screen group. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pusScreenGrp* parameter has the following form:

```
struct {  
    USHORT idScreenGrp;  
    USHORT fTerminate;  
};
```

Field	Description
idScreenGrp	Specifies the screen-group identifier of the new foreground screen. It must be a value in the range 0 to 15.
fTerminate	Specifies whether the screen group is terminating. If it is 0x0000, the screen group is switching. If it is 0xFFFF, the screen group is terminating.

- **int DosDevIOCtl(0L, *phkbd*, 0x0057, 0x0004, *hDevice*)**
PHKBD *phkbd*; pointer to variable with logical keyboard handle
HFILE *hDevice*; device handle

The SETFOCUS function sets the keyboard focus to the specified logical keyboard.

Parameter	Description
<i>phkbd</i>	Points to a logical keyboard handle. The handle must have been previously created using the KbdOpen function.

hDevice Identifies the keyboard device that will receive the device-control function. The handle must have been previously created using the **DosOpen** function.

Return Value

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

■ **int DosDevIOCtl**(*pbFrameCtl*, *pbCommand*, **0x0042**, **0x0005**, *hDevice*)
PBYTE *pbFrameCtl*; pointer to buffer with frame settings
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The SETFRAMECTL function sets frame control for a printer device.

Parameter	Description
<i>pbFrameCtl</i>	Points to the structure containing the frame-control information. For a full description, see the following "Structures" section.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the printer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbFrameCtl* parameter has the following form:

```
struct {  
    BYTE bCharsPerLine;  
    BYTE bLinesPerInch;  
};
```

Field	Description
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

GETFRAMECTL

- **int DosDevIOCtl(0L, *pfHotKey*, 0x0055, 0x0007, *hDevice*)**
PUSHORT *pfHotKey*; pointer to variable with hot key
HFILE *hDevice*; device handle

The SETHOTKEYBUTTON function sets the mouse-button equivalent for the system hot key.

Parameter	Description										
<i>pfHotKey</i>	Points to the variable specifying the hot-key button. It can be one or more of the following values:										
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0001</td><td>No system hot key desired.</td></tr> <tr> <td>MHK_BUTTON1</td><td>Button 1 is system hot key.</td></tr> <tr> <td>MHK_BUTTON2</td><td>Button 2 is system hot key.</td></tr> <tr> <td>MHK_BUTTON3</td><td>Button 3 is system hot key.</td></tr> </table>	Value	Meaning	0x0001	No system hot key desired.	MHK_BUTTON1	Button 1 is system hot key.	MHK_BUTTON2	Button 2 is system hot key.	MHK_BUTTON3	Button 3 is system hot key.
Value	Meaning										
0x0001	No system hot key desired.										
MHK_BUTTON1	Button 1 is system hot key.										
MHK_BUTTON2	Button 2 is system hot key.										
MHK_BUTTON3	Button 3 is system hot key.										
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.										

Return Value

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

Comments

If 0x0001 is specified, no system hot-key support is provided. If multiple values are given, not including 0x0001, the system hot key is interpreted as requiring the indicated buttons to be pressed simultaneously.

This function may only be called by the process that initially issues it. It should be used only by the command shell.

- **int DosDevIOCtl**(*pfRetry*, *pbCommand*, **0x0044**, **0x0005**, *hDevice*)
PBYTE *pfRetry*; pointer to retry flag
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The SETINFINITERETRY function sets infinite retry for a print device.

Parameter	Description
<i>pfRetry</i>	Points to a variable that specifies whether to enable infinite retry. If the variable is 0x0000, the function disables infinite retry. If it is 0x0001, the function enables infinite retry.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the printer device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

GETINFINITERETRY

- **int DosDevIOCtl**(**0L**, *pbInputMode*, **0x0051**, **0x0004**, *hDevice*)
PBYTE *pbInputMode*; pointer to variable with input mode
HFILE *hDevice*; device handle

The SETINPUTMODE function sets the input-and-shift reporting modes for the keyboard device driver. The input mode defines whether the following input keys are processed as keystrokes or commands: CONTROL+C, CONTROL+BREAK, CONTROL+S, CONTROL+P, SCROLL LOCK, PRINTSCREEN.

The shift-reporting mode defines whether the shift keys are processed as shift keys or keystrokes.

Parameter	Description
<i>pbInputMode</i>	Points to a variable that contains the input mode for the keyboard. If it is 0x00, the function sets the cooked-input mode. If it is 0x80, the function sets the raw-input mode. If these values are combined with 0x01, the function enables shift-reporting mode. Otherwise, the mode is disabled.
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Comments

The default input mode is cooked. The keyboard device driver maintains an input mode for each screen group.

See Also

GETINPUTMODE

■ **int DosDevIOCtl(OL, *pfFlags*, 0x0052, 0x0004, *hDevice*)**
PBYTE *pfFlags*; pointer to variable with flags
HFILE *hDevice*; device handle

The SETINTERIMFLAG function sets the interim character flags.

Parameter	Description						
<i>pfFlags</i>	Points to the variable that contains the interim flags. The variable can be one or both of the following values:						
	<table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0020</td><td>Conversion requested.</td></tr> <tr> <td>0x0080</td><td>Interim character flag on.</td></tr> </table>	Value	Meaning	0x0020	Conversion requested.	0x0080	Interim character flag on.
Value	Meaning						
0x0020	Conversion requested.						
0x0080	Interim character flag on.						
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.						

Return Value

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

Comments

The keyboard device driver maintains the interim character flags for each screen group. The keyboard device driver 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.

- **USHORT DosDevIOCtrl(0L, *pbLineCtrl*, 0x0042, 0x0001, *hDevice*)**
PBYTE *pbLineCtrl*; pointer to buffer with line settings
HFILE *hDevice*; device handle

The SETLINECTRL function sets the line characteristics (stop bits, parity, and data bits) for the specified serial device.

Parameter	Description
<i>pbLineCtrl</i>	Points to a structure that contains the settings for the number of data bits, parity, and number of stop bits. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. The function returns a "general failure" error code (ERROR_GEN_FAILURE) if any one of the specified line characteristics is out of range. When an error occurs, line characteristics remain unchanged.

Structures

The structure pointed to by the *pbLineCtrl* parameter has the following form:

```
struct {  
    BYTE bDataBits;  
    BYTE bParity;  
    BYTE bStopBits;  
};
```

Field	Description												
bDataBits	Specifies the number of data bits to be used. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x05</td><td>5 data bits</td></tr><tr><td>0x06</td><td>6 data bits</td></tr><tr><td>0x07</td><td>7 data bits</td></tr><tr><td>0x08</td><td>8 data bits</td></tr></table>	Value	Meaning	0x05	5 data bits	0x06	6 data bits	0x07	7 data bits	0x08	8 data bits		
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: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x00</td><td>No parity</td></tr><tr><td>0x01</td><td>Odd parity</td></tr><tr><td>0x02</td><td>Even parity</td></tr><tr><td>0x03</td><td>Mark parity (parity bit is always 1)</td></tr><tr><td>0x04</td><td>Space parity (parity bit is always 0)</td></tr></table>	Value	Meaning	0x00	No parity	0x01	Odd parity	0x02	Even parity	0x03	Mark parity (parity bit is always 1)	0x04	Space parity (parity bit is always 0)
Value	Meaning												
0x00	No parity												
0x01	Odd parity												
0x02	Even parity												
0x03	Mark parity (parity bit is always 1)												
0x04	Space parity (parity bit is always 0)												
bStopBits	Specifies the number of stop bits used. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x00</td><td>1 stop bit</td></tr><tr><td>0x01</td><td>1.5 stop bits (only valid with 5-bit word length)</td></tr><tr><td>0x02</td><td>2 stop bits (not valid with 5-bit word length)</td></tr></table>	Value	Meaning	0x00	1 stop bit	0x01	1.5 stop bits (only valid with 5-bit word length)	0x02	2 stop bits (not valid with 5-bit word length)				
Value	Meaning												
0x00	1 stop bit												
0x01	1.5 stop bits (only valid with 5-bit word length)												
0x02	2 stop bits (not valid with 5-bit word length)												

Comments

When a device is first opened, the initial line characteristics are 7 data bits, even parity, and 1 stop bit. After the line characteristics are changed, they remain unchanged 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, but 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 will not recognize the XOFF character even

when automatic transmit control is enabled. On the other hand, if the error substitution character is 0x80, the device driver still places 0x80 in the receive queue. Programs are responsible for making sure these characters match the specified data size. Any characters that were in the receive queue before the function is called remain unchanged.

See Also

GETLINECTRL

- **int DosDevIOCtl**(*pbDrive*, *pbCommand*, **0x0003**, **0x0008**, *hDevice*)
PBYTE *pbDrive*; pointer to variable with drive
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The SETLOGICALMAP function sets the logical-drive mapping for a block device.

Parameter	Description
<i>pbDrive</i>	Points to a variable that contains the logical-drive number. The number 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 onto the physical drive, the function sets the variable to zero.
<i>pbCommand</i>	Points to a variable containing a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

GETLOGICALMAP

USHORT	DosDevIOCtl	(<i>pfCommErr</i> , <i>pbCtrlSignals</i> , 0x0046 , 0x0001 , <i>hDevice</i>)
PUSHORT	<i>pfCommErr</i> ;	pointer to variable for error value
PBYTE	<i>pbCtrlSignals</i> ;	pointer to buffer with control signals
HFILE	<i>hDevice</i> ;	device handle

The **SETMODEMCTRL** function sets the modem-control signals. The 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.

Parameter	Description										
<i>pfCommErr</i>	Points to the variable that receives the communication status of the device. It can be a combination of the following values:										
	<table border="1"> <thead> <tr> <th>Value</th><th>Meaning</th></tr> </thead> <tbody> <tr> <td>0x0001</td><td>Receive queue overrun. There is no room in the device driver receive queue to put a character read in from the receive hardware.</td></tr> <tr> <td>0x0002</td><td>Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.</td></tr> <tr> <td>0x0004</td><td>The hardware detected a parity error.</td></tr> <tr> <td>0x0008</td><td>The hardware detected a framing error.</td></tr> </tbody> </table> <p>The function sets the variable to zero if it encounters an error.</p>	Value	Meaning	0x0001	Receive queue overrun. There is no room in the device driver receive queue to put a character read in from the receive hardware.	0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.	0x0004	The hardware detected a parity error.	0x0008	The hardware detected a framing error.
Value	Meaning										
0x0001	Receive queue overrun. There is no room in the device driver receive queue to put a character read in from the receive hardware.										
0x0002	Receive hardware overrun. A character arrived before the previous character was completely read. The previous character is lost.										
0x0004	The hardware detected a parity error.										
0x0008	The hardware detected a framing error.										
<i>pbCtrlSignals</i>	Points to the structure that contains the settings for the modem-control signals. For a full description, see the following “Structures” section.										
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.										

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (`ERROR_GEN_FAILURE`) if the specified signal settings are invalid. When an error occurs, the signal settings remain unchanged.

Structures

The structure pointed to by the *pbCtrlSignals* parameter has the following form:

```
struct {  
    BYTE fbModemOn;  
    BYTE fbModemOff;  
};
```

Field	Description								
fbModemOn	Specifies the modem-control signals to be enabled. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x01</td><td>Enable the data-terminal-ready (DTR) signal.</td></tr><tr><td>0x02</td><td>Enable the ready-to-transmit (RTS) signal.</td></tr><tr><td colspan="2">If the field is 0x00, no signals are enabled.</td></tr></table>	Value	Meaning	0x01	Enable the data-terminal-ready (DTR) signal.	0x02	Enable the ready-to-transmit (RTS) signal.	If the field is 0x00, no signals are enabled.	
Value	Meaning								
0x01	Enable the data-terminal-ready (DTR) signal.								
0x02	Enable the ready-to-transmit (RTS) signal.								
If the field 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: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0xFE</td><td>Disable the data-terminal-ready (DTR) signal.</td></tr><tr><td>0xFD</td><td>Disable the ready-to-transmit (RTS) signal.</td></tr><tr><td colspan="2">If the field is 0xFF, no signals are disabled.</td></tr></table>	Value	Meaning	0xFE	Disable the data-terminal-ready (DTR) signal.	0xFD	Disable the ready-to-transmit (RTS) signal.	If the field is 0xFF, no signals are disabled.	
Value	Meaning								
0xFE	Disable the data-terminal-ready (DTR) signal.								
0xFD	Disable the ready-to-transmit (RTS) signal.								
If the field is 0xFF, no signals are disabled.									

Any values other than those specified for the **bModemOn** and **bModemOff** fields will cause a “general failure” (ERROR_GEN_FAILURE) error code.

Comments

The 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 SETDCBINFO function (0x0001,0x0053).

After a serial device has been closed, the device driver turns the DTR and RTS signals off, 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

GETMODEMOUTPUT

■ **int DosDevIOCtl(0L, *pfStatus*, 0x005C, 0x0007, *hDevice*)**
PUSHORT *pfStatus*; pointer to variable with status
HFILE *hDevice*; device handle

The SETMOUSTATUS function sets a subset of the current mouse device-driver status flags.

Parameter	Description						
<i>pfStatus</i>	Points to the variable that contains the status flags for the input device. It can be one or both of the following values: <table> <tr> <th>Value</th><th>Meaning</th></tr> <tr> <td>0x0100</td><td>Interrupt-level pointer-draw routine not called.</td></tr> <tr> <td>0x0200</td><td>Mouse data being returned in mickeys (not pixels).</td></tr> </table>	Value	Meaning	0x0100	Interrupt-level pointer-draw routine not called.	0x0200	Mouse data being returned in mickeys (not pixels).
Value	Meaning						
0x0100	Interrupt-level pointer-draw routine not called.						
0x0200	Mouse data being returned in mickeys (not pixels).						
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.						

Return Value

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

- **int DosDevIOCtl(0L, *pbFunction*, 0x005A, 0x0007, *hDevice*)**
PBYTE *pbFunction*; pointer to buffer with draw function
HFILE *hDevice*; device handle

The 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.

Parameter	Description
<i>pbFunction</i>	Points to the structure containing the address of the pointer-draw function. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbFunction* parameter has the following form:

```
struct {  
    PFN pfnDraw;  
    PCH pchDataSeg;  
};
```

Field	Description
pfnDraw	Specifies the address of the pointer-draw function.
pchDataSeg	Specifies the starting address of the data segment of the pointer-draw function.

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.

The mouse device driver issues a **FAR** call to the pointer-draw device driver whenever a mouse interrupt occurs that requires action concerning the pointer image. In addition, the mouse device driver may call the pointer-draw routine as a result of some action on the part of the application, such as when the **MouDrawPtr**, **MouRemovePtr**, **MouSetPtrPos**, **MouSetPtrShape**, or **MouGetPtrShape** function is called.

■ **int DosDevIOCtl(0L, *pplPosition*, 0x0059, 0x0007, *hDevice*)**
PPTRLOC *pplPosition*; pointer to buffer with position
HFILE *hDevice*; device handle

The SETPTRPOS function assigns a new screen position for the pointer image.

Parameter	Description
<i>pplPosition</i>	Points to the structure that contains the new position for the pointer.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pplPosition* parameter has the following form:

```
typedef struct _PTRLOC {
    USHORT row;
    USHORT col;
} PTRLOC;
```

Field	Description
row	Specifies the <i>x</i> -coordinate of the pointer.
col	Specifies the <i>y</i> -coordinate of the pointer.

Comments

The coordinate values depend on the display mode. Character-position values must be used if the display is in text mode. Pixel values must be used if the display is in graphics mode.

This 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 will remain hidden (invisible) until either enough mouse movement is generated to place it outside of the exclusion rectangle, or until the exclusion rectangle is released.

■ **int** DosDevIOCtl(*pbBuffer*, *ppsShape*, **0x0056**, **0x0007**, *hDevice*)
PBYTE *pbBuffer*; pointer to buffer with shape masks
PPTRSHAPE *ppsShape*; pointer to buffer with shape info
HFILE *hDevice*; device handle

The SETPTRSHAPE function sets the shape of the pointer.

Parameter	Description
<i>pbBuffer</i>	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.
<i>ppsShape</i>	Points to the structure that contains the pointer information and shape. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *ppsShape* parameter has the following form:

```
typedef struct _PTRSHAPE {
    USHORT cb;
    USHORT col;
    USHORT row;
    USHORT colHot;
    USHORT rowHot;
} PTRSHAPE;
```

Field	Description
cb	Specifies the length in bytes of the pointer shape buffer. This should be equal to 4 (for text mode) and $\text{row} \times \text{col} \times \text{bits-per-pixel} \times 2/8$ (for graphics mode).
col	Specifies the width in columns of the pointer image.
row	Specifies the height in rows of the pointer image.
colHot	Specifies the offset in columns from the first column to the pointer hot spot.
rowHot	Specifies the offset in rows from the first row to the pointer hot spot.

For text mode, the **row** and **col** fields must be set to 1. For graphics mode, these must be greater than 1.

Comments

All the pointer-definition record fields and the pointer-image buffers are validated using the screen group's mode table values. The parameter values must be the same mode as the current screen group's display mode. For text mode, these values must be character values; for graphics mode, they must be pixel values.

■ **int DosDevIOctl(0L, *pbFunction*, 0x005B, 0x0007, *hDevice*)**
PBYTE *pbFunction*; pointer to buffer with function
HFILE *hDevice*; device handle

The SETREALDRAWADDRESS function notifies the real-mode mouse device driver of the entry point of a real-mode pointer-draw routine. The function is intended for use by the shell (session manager) at the end of system initialization. The function is valid for real mode only.

SETREALDRAWADDRESS

Parameter	Description
<i>pbFunction</i>	Points to the structure that contains the address of the pointer-draw function. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbFunction* parameter has the following form:

```
struct {  
    PFN pfnDraw;  
    PCH pchDataSeg;  
};
```

Field	Description
pfnDraw	Specifies the address of the pointer-draw function.
pchDataSeg	Specifies the starting address of the data segment of the pointer-draw function.

See Also

SETPROTDRAWADDRESS

int DosDevIOCtl(OL, *psfFactors*, 0x0053, 0x0007, *hDevice*)
PSCALEFACT *psfFactors*; pointer to buffer with factors
HFILE *hDevice*; device handle

The 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 will report a mouse event. In graphics mode, the ratio is mickeys per pixel. In text mode, the ratio is mickeys per character. By default, the ratio values are one mickey-per-row unit and one mickey-per-column unit.

Parameter	Description
<i>psfFactors</i>	Points to the structure that contains the scaling factors. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *psfFactors* parameter has the following form:

```
typedef struct _SCALEFACT {
    USHORT rowScale;
    USHORT colScale;
} SCALEFACT;
```

Field	Description
rowScale	Specifies the horizontal coordinate scaling factor. It must be a value from 0 to 32,767.
colScale	Specifies the vertical coordinate scaling factor. It must be a value from 0 to 32,767.

- int DosDevIOCtl(0L, *pbHotKey*, 0x0056, 0x0004, *hDevice*)**
PBYTE *pbHotKey*; pointer to buffer with hot key
HFILE *hDevice*; device handle

The SETSESMGRHOTKEY function sets the session-manager hot keys. This function is intended to be used by the session manager to set a list of keyboard hot keys. A new hot key applies to all screen groups. Up to sixteen hot keys can be defined by the session manager.

Parameter	Description
<i>pbHotKey</i>	Points to the structure that contains the hot-key information. For a full description, see the the “Structures” section.

hDevice Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the **DosOpen** function.

Return Value

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

Structures

The structure pointed to by the *pbHotKey* parameter has the following form:

```
struct {
    USHORT fHotKey;
    UCHAR  scancodeMake;
    UCHAR  scancodeBreak;
    USHORT idHotKey;
};
```

Field	Description																						
fHotKey	Specifies the setting for the session-manager hot key. It can be a combination of the following values:																						
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>Right SHIFT key down.</td></tr><tr><td>0x0002</td><td>Left SHIFT key down.</td></tr><tr><td>0x0100</td><td>Left CONTROL key down.</td></tr><tr><td>0x0200</td><td>Left ALT key down.</td></tr><tr><td>0x0400</td><td>Right CONTROL key down.</td></tr><tr><td>0x0800</td><td>Right ALT key down.</td></tr><tr><td>0x1000</td><td>SCROLL LOCK key down.</td></tr><tr><td>0x2000</td><td>NUM LOCK key down.</td></tr><tr><td>0x4000</td><td>CAPS LOCK key down.</td></tr><tr><td>0x8000</td><td>SYSREQ key down.</td></tr></table>	Value	Meaning	0x0001	Right SHIFT key down.	0x0002	Left SHIFT key down.	0x0100	Left CONTROL key down.	0x0200	Left ALT key down.	0x0400	Right CONTROL key down.	0x0800	Right ALT key down.	0x1000	SCROLL LOCK key down.	0x2000	NUM LOCK key down.	0x4000	CAPS LOCK key down.	0x8000	SYSREQ key down.
Value	Meaning																						
0x0001	Right SHIFT key down.																						
0x0002	Left SHIFT key down.																						
0x0100	Left CONTROL key down.																						
0x0200	Left ALT key down.																						
0x0400	Right CONTROL key down.																						
0x0800	Right ALT key down.																						
0x1000	SCROLL LOCK key down.																						
0x2000	NUM LOCK key down.																						
0x4000	CAPS LOCK key down.																						
0x8000	SYSREQ key down.																						
scancodeMake	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.																						

- scancodeBreak** 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 to 15.

The **scancodeMake** and **scancodeBreak** fields are mutually exclusive; either may be specified, but not both. The use of either indicates when to recognize the hot key.

Comments

The SETSESMGRHOTKEY function is successful only if it is performed by the process that initially called the 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

SETFGNDSCREENGRP

- **int DosDevIOCtl(0L, *pbShiftState*, 0x0053, 0x0004, *hDevice*)**
PBYTE *pbShiftState*; pointer to buffer with shift state
HFILE *hDevice*; device handle

The SETSHIFTSTATE control function sets the shift state for the keyboard. The shift state identifies the state of the shift keys; that is, whether the SHIFT, CONTROL, ALT, INSERT, and SYSREQ keys are up or down and whether the SCROLL LOCK, NUMLOCK, and CAPSLOCK keys are toggled.

The system puts the shift state into the character data record built for each incoming keystroke, so the shift state 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.

Parameter	Description
<i>pbShiftState</i>	Points to the structure that contains the shift state. For a full description, see the following “Structures” section.

SETSHIFTSTATE

hDevice Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the **DosOpen** function.

Return Value

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

Structures

The structure pointed to by the *pbShiftState* parameter has the following form:

```
struct {  
    USHORT fState;  
    BYTE fNLS;  
};
```

Field	Description
fState	Specifies shift states. It can be any combination of the following values:
Value	Meaning
0x0001	Right SHIFT key down.
0x0002	Left SHIFT key down.
0x0004	Either CONTROL key down.
0x0008	Either ALT key down.
0x0010	SCROLL LOCK mode on.
0x0020	NUMLOCK mode on.
0x0040	CAPSLOCK mode on.
0x0080	INSERT mode on.
0x0100	Left CONTROL key down.
0x0200	Left ALT key down.
0x0400	Right CONTROL key down.
0x0800	Right ALT key down.
0x1000	SCROLL LOCK key down.
0x2000	NUMLOCK key down.
0x4000	CAPSLOCK key down.
0x8000	SYSREQ key down.

fNLS Specifies the national-language-dependent shift states.
This byte is zero for the United States.

Comments

The keyboard device driver maintains a shift state for each screen group.

See Also

GETSHIFTSTATE

■ **int DosDevIOCtl(0L, *pbTransTable*, 0x0050, 0x0004, *hDevice*)**
PBYTE *pbTransTable*; pointer to translation table
HFILE *hDevice*; device handle

The SETTRANSTABLE control function passes a new scancode-to-character translation table to the keyboard translation function. The new table, which overlays the table currently in use, translates subsequent keystrokes.

Parameter	Description
<i>pbTransTable</i>	Points to a translation table. For a full description, see Appendix C, "Code Pages."
<i>hDevice</i>	Identifies the keyboard device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Comments

The default translate table is U.S. English.

SETTYPAMATICRATE

■ **int** DosDevIOCtl(**0L**, *pusRateDelay*, **0x0054**, **0x0004**, *hDevice*)
PUSHORT *pusRateDelay*; typeamatic rate and delay
HFILE *hDevice*; device handle

The SETTYPAMATICRATE function sets the keyboard typeamatic rate and delay to the values specified in the request.

Parameter	Description
<i>pusRateDelay</i>	Points to the structure that contains the typeamatic rate and delay. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the keyboard device that will receive the device-control function. The handle must have been previously created using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pusRateDelay* parameter has the following form:

```
struct {  
    USHORT delay;  
    USHORT rate;  
};
```

Field	Description
delay	Specifies the typeamatic delay in milliseconds. A value greater than the maximum value defaults to the maximum value.
rate	Specifies the typeamatic rate in characters per second. A value greater than the maximum value defaults to the maximum value.

See Also

GETTYPAMATICRATE

- **USHORT DosDevIOCtl(0L, 0L, 0x0048, 0x0001, *hDevice*)**
HFILE *hDevice*; device handle

The STARTTRANSMIT function starts transmission. It is similar to the device receiving the XON character. This function allows data transmission to be resumed by the device driver if data transmission is halted due to a STOPTRANSMIT function (0x0001,0x0047) or due to an XOFF character being received while the device driver is in automatic transmit flow control mode.

Parameter	Description
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (ERROR_GEN_FAILURE) when an error occurs.

Comments

There may be other reasons why transmission may be disabled; transmission may not be resumed. See the GETCOMMSTATUS function (0x0001,0x0064).

See Also

GETCOMMSTATUS, STOPTRANSMIT

- **USHORT DosDevIOCtl(0L, 0L, 0x0047, 0x0001, *hDevice*)**
HFILE *hDevice*; device handle

The STOPTRANSMIT function stops the device from transmitting. It is similar to the device receiving the XOFF character. This function stops data transmission by preventing the device driver from sending additional data to the transmit hardware.

Parameter	Description
<i>hDevice</i>	Identifies the serial device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

The return value is zero if the function is successful. The function returns a “general failure” error code (`ERROR_GEN_FAILURE`) when an error occurs.

If automatic transmit flow control is enabled, this request causes the device driver to behave exactly as if it received the XOFF character. Transmission can be resumed (due to being stopped from this request) when an XON character is received by the device driver, when a START-TRANSMIT (0x0001,0x0048) function is received, or when the device driver is told to disable automatic transmit flow control and in the previous state automatic transmit flow control was enabled.

There still may be other reasons why transmission may be disabled. See the GETCOMMSTATUS function (0x0001,0x0064).

GETCOMMSTATUS, STARTTRANSMIT

The TRANSMITIMM function transmits the specified byte immediately.

Return Value

The return value is zero if the function is successful. The function returns a “general failure” error code (`ERROR_GEN_FAILURE`) when 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 receive flow 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 `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 transmit or receive flow control (XON/XOFF) being enabled or due to being asked to behave as if an XOFF character had been received, the device driver will still transmit a character that is waiting to be transmitted immediately due to this request. An application which requests that the device driver transmit a character immediately when automatic transmit or receive flow 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 receive flow control may or may not be placed ahead of the character waiting to be transmitted immediately. Applications should not have dependencies on this ordering.

UNLOCKDRIVE

- **int DosDevIOCtl(0L, *pbCommand*, 0x0008, 0x0001, *hDevice*)**
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The UNLOCKDRIVE function unlocks a drive. The locked volume represented by the handle is required in the drive.

Parameter	Description
<i>pbCommand</i>	Points to a variable that contains a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

LOCKDRIVE

- **int DosDevIOCtl(0L, *pbCommand*, 0x0001, 0x0009, *hDevice*)**
PBYTE *pbCommand*; pointer to variable with command
HFILE *hDevice*; device handle

The UNLOCKPHYSDRIVE function unlocks the physical disk drive and any of its associated logical units. The function also affects the logical units on the physical disk drive.

Parameter	Description
<i>pbCommand</i>	Points to a variable that contains a reserved value. The value must be zero.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

See Also

LOCKPHYSDRIVE

■ **int** DosDevIOCtl(**0L**, *pvmiMode*, **0x0051**, **0x0007**, *hDevice*)
PVIOMODEINFO *pvmiMode*; pointer to buffer with screen mode
HFILE *hDevice*; device handle

The UPDATEDISPLAYMODE function notifies the mouse device driver that the screen display mode has been modified.

Parameter	Description
<i>pvmiMode</i>	Points to the structure that contains the video-mode information. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the mouse device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pvmiMode* parameter 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;
```

Field	Description
cb	Specifies the length in bytes of the structure.
fbType	Specifies the video-mode characteristics. It can be a combination of the following values:

	Value	Meaning
	0x0001	Set adapter to other than a monochrome/printer adapter. If this field is not given, the monochrome/printer adapter is assumed by default.
	0x0002	Set graphics mode. If this field is not given, the adapter is set to text mode.
	0x0004	Disable color-burst mode. If this field is not given, color burst mode is enabled.
color	Specifies the number of colors. It can be one of the following values:	
	Value	Meaning
	0x0001	2 colors
	0x0002	4 colors
	0x0004	16 colors
col	Specifies the number of alphanumeric columns.	
row	Specifies the number of alphanumeric rows.	
hres	Specifies the number of pixels per column.	
vres	Specifies the number of pixels per row.	
fmt_ID	Reserved. Set to zero.	
attrib	Reserved. Set to zero.	

Comments

Whenever the video I/O subsystem or registered video I/O subsystem sets or resets the display mode, it must synchronize the mouse device driver's pointer-update functions by providing this notification record to the mouse driver prior to switching display modes.

See Also

VioSetMode

- int DosDevIOctl(0L, pbCommand, 0x0065, 0x0009, hDevice)**
PBYTE pbCommand; pointer to buffer with command
HFILE hDevice; device handle

The VERIFYPHYSTRACK function verifies input/output operations on a physical track. The function performs the verify operation on the device

specified in the request. Its operation is similar to the 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 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 verified.

Parameter	Description
<i>pbCommand</i>	Points to the structure that contains information about the verify operation. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the physical device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbCommand* parameter has the following form:

```
struct {
    BYTE    bCommand;
    USHORT  head;
    USHORT  cylinder;
    USHORT  firstSector;
    USHORT  cSectors;
    struct {
        USHORT  sectorNumber;
        USHORT  sectorSize;
    } TrackTable[N];
};
```

Field	Description
bCommand	Specifies the type of track layout. If it 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.

head	Specifies the physical head on the disk drive on which to perform the verify operation.
cylinder	Specifies the cylinder number on which to perform the verify operation.
firstSector	Specifies the logical sector number at which to start the verify 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 to verify, up to the maximum specified in the track-layout table. The function does not step heads and tracks.
TrackTable[N]	Specifies the track-layout table. It is an array of structures that contain the numbers and sizes of the sectors in the track. The sectorNumber field specifies the sector number. The sectorSize field specifies the size of the sector. The first element defines the sector number and size in bytes of the first sector on the track. The number of elements depends on the number of sectors on the track.

See Also

READPHYSTRACK, WRITEPHYSTRACK, VERIFYTRACK

■ **int DosDevIOCtl(0L, *pbCommand*, 0x0065, 0x0008, *hDevice*)**
PBYTE *pbCommand*; pointer to buffer with command
HFILE *hDevice*; device handle

The VERIFYTRACK function verifies an operation on a drive. The function performs the verify operation on the device specified in this request. The track-layout table passed in the function determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is verified.

Parameter	Description
<i>pbCommand</i>	Points to the structure that contains information about the verify operation. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbCommand* parameter has the following form:

```
struct {
    BYTE    bCommand;
    USHORT  head;
    USHORT  cylinder;
    USHORT  firstSector;
    USHORT  cSectors;
    struct {
        USHORT sectorNumber;
        USHORT sectorSize;
    } TrackTable[N];
};
```

Field	Description
bCommand	Specifies the type of track layout. If it 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.
head	Specifies the physical head on the drive on which to perform the verify operation.
cylinder	Specifies the cylinder number on which to perform the verify operation.
firstSector	Specifies the logical sector number at which to start the verify 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 to verify, up to the maximum specified in the track-layout table. The function does not step heads and tracks.
TrackTable[N]	Specifies the track-layout table. It is an array of structures that contain the numbers and sizes of the sectors in the track. The sectorNumber field specifies the sector number. The sectorSize field specifies the size of the sector. The first element defines the sector number and size in bytes of the first sector on the track. The number of elements depends on the number of sectors on the track.

See Also

READTRACK, WRITETRACK

- **int DosDevIOCtl(*pbBuffer*, *pbCommand*, 0x0044, 0x0009, *hDevice*)**
PBYTE *pbBuffer*; pointer to buffer with data
PBYTE *pbCommand*; pointer to buffer with command
HFILE *hDevice*; device handle

The WRITEPHYSTRACK function writes to a physical track. The function performs the write operation on the device specified in the request. Its operation is similar to that of the 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 the function determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is written.

Parameter	Description
<i>pbBuffer</i>	Points to the buffer that contains the data to be written.
<i>pbCommand</i>	Points to the structure that contains information about the write operation. For a full description, see the following "Structures" section.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbCommand* parameter has the following form:

```
struct {  
    BYTE    bCommand;  
    USHORT  head;  
    USHORT  cylinder;  
    USHORT  firstSector;  
    USHORT  cSectors;  
    struct {  
        USHORT  sectorNumber;  
        USHORT  sectorSize;  
    } TrackTable[N];  
};
```

Field	Description
bCommand	Specifies the type of track layout. If it 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.
head	Specifies the physical head on the disk drive on which to perform the write operation.
cylinder	Specifies the cylinder number on which to perform the write operation.
firstSector	Specifies the logical sector number at which to start the write 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 to write, up to the maximum specified in the track-layout table. The function does not step heads and tracks.
TrackTable[N]	Specifies the track-layout table. It is an array of structures that contain the numbers and sizes of the sectors in the track. The sectorNumber field specifies the sector number. The sectorSize field specifies the size of the sector. The first element defines the sector number and size in bytes of the first sector on the track. The number of elements depends on the number of sectors on the track.

See Also

READPHYSTRACK, WRITETRACK

- **int DosDevIOctl**(*pbBuffer*, *pbCommand*, **0x0044**, **0x0008**, *hDevice*)
PBYTE *pbBuffer*; pointer to buffer with data
PBYTE *pbCommand*; pointer to buffer with command
HFILE *hDevice*; device handle

The WRITETRACK function writes to a track on a drive. The function performs the write operation to the device specified in this request. The track-layout table passed in the function determines the sector number, which is passed to the disk controller for the operation. When the sectors are oddly numbered or nonconsecutive, the request is broken into an appropriate number of single-sector operations, and one sector at a time is written.

Parameter	Description
<i>pbBuffer</i>	Points to the buffer that contains the data to be written.
<i>pbCommand</i>	Points to the structure that contains information about the write operation. For a full description, see the following “Structures” section.
<i>hDevice</i>	Identifies the disk-drive device that receives the device-control function. The handle must have been previously created by using the DosOpen function.

Return Value

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

Structures

The structure pointed to by the *pbCommand* parameter has the following form:

```
struct {
    BYTE    bCommand;
    USHORT  head;
    USHORT  cylinder;
    USHORT  firstSector;
    USHORT  cSectors;
    struct {
        USHORT  sectorNumber;
        USHORT  sectorSize;
    } TrackTable[N];
};
```

Field	Description
bCommand	Specifies the type of track layout. If it 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.
head	Specifies the physical head on the disk drive on which to perform the write operation.
cylinder	Specifies the cylinder number on which to perform the write operation.
firstSector	Specifies the logical sector number at which to start the write 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 to write, up to the maximum specified in the track-layout table. The function does not step heads and tracks.
TrackTable[N]	Specifies the track-layout table. It is an array of structures that contain the numbers and sizes of the sectors in the track. The sectorNumber field specifies the sector number. The sectorSize field specifies the size of the sector. The first element defines the sector number and size in bytes of the first sector on the track. The number of elements depends on the number of sectors on the track.

See Also

READTRACK

Appendixes

Appendix A: Utility Macros	517
Appendix B: Devices	525
Appendix C: Code Pages	531
Appendix D: Format of MS OS/2 Executable Files	555
Appendix E: ANSI Escape Sequences	569

1

2

3

Appendix A

Utility Macros

A.1	Introduction	519
A.2	Macros	519

A.1 Introduction

This appendix defines the utility macros provided with the MS OS/2 C-language header files. The utility macros are used with MS OS/2 functions to create structures, pointers, and type conversions quickly. The following are the MS OS/2 utility macros:

DEFINEMUXSEMLIST	LOUSHORT	MAKETYPE
FIELDOFFSET	MAKEERRORID	MAKEULONG
HIBYTE	MAKELONG	MAKEUSHORT
HIUCHAR	MAKEP	OFFSETOF
HIUSHORT	MAKEPGINFOSEG	SELECTOROF
LOBYTE	MAKEPLINFOSEG	
LOUCHAR	MAKESHORT	

A.2 Macros

This section lists the MS OS/2 utility macros, gives each macro’s purpose, and describes its parameters.

■

DEFINEMUXSEMLIST(*name*, *size*)

This macro creates a structure used to hold the semaphore list for the **DosMuxSemWait** function. The structure receives a name specified by the *name* parameter and is set to the size given by the *size* parameter.

Parameter	Description
<i>name</i>	Specifies the name of the structure to be created.
<i>size</i>	Specifies the size of the structure; that is, the number of semaphores in the list.

See Also

DosMuxSemWait

■ FIELDOFFSET(*type*, *field*)

This macro computes the address offset of the specified field in the structure specified by the *type* parameter.

Parameter	Description
<i>type</i>	Specifies the name of the structure.
<i>field</i>	Specifies the name of a field defined within the given structure.

■ HIBYTE(*w*)

This macro retrieves the high-order byte of the 16-bit value specified by the *w* parameter.

Parameter	Description
<i>w</i>	Specifies a 16-bit value.

■ HIUCHAR(*w*)

This macro retrieves the high-order byte of the 16-bit value specified by the *w* parameter.

Parameter	Description
<i>w</i>	Specifies a 16-bit value.

■ HIUSHORT(*l*)

This macro retrieves the high-order 16-bit word from the 32-bit value specified by the *l* parameter.

Parameter	Description
<i>l</i>	Specifies a 32-bit value.

■ LOBYTE(*w*)

This macro retrieves the low-order 8-bit byte from the 16-bit value specified by the *w* parameter.

Parameter	Description
<i>w</i>	Specifies a 16-bit value.

■ LOUCHAR(*w*)

This macro retrieves the low-order 8-bit byte from the 16-bit value specified by the *w* parameter.

Parameter	Description
<i>w</i>	Specifies a 16-bit value.

■ LOUSHORT(*l*)

This macro retrieves the low-order 16-bit word from the 32-bit value specified by the *l* parameter.

Parameter	Description
<i>l</i>	Specifies a 32-bit value.

■ MAKEERRORID(*sev*, *error*)

This macro creates an error identifier that consists of a severity level and an error value. Error identifiers are intended to be used with a later release of MS OS/2.

Parameter	Description
<i>sev</i>	Specifies a severity level. It can be any value from 0 to 65,535.
<i>error</i>	Specifies an error value. It can be any value from 0 to 65,535.

■ MAKELONG(*l*, *h*)

This macro combines two 16-bit word values to create a 32-bit long integer.

Parameter	Description
<i>l</i>	Specifies the low-order 16-bit value for the new integer.
<i>h</i>	Specifies the high-order 16-bit value for the new integer.

■ MAKEP(*sel*, *off*)

This macro combines a segment selector and an address offset to create a far (32-bit) pointer to a memory address.

Parameter	Description
<i>sel</i>	Specifies a segment selector. It must be a valid segment selector, for example, it may have been previously created by using the DosAllocSeg function.
<i>off</i>	Specifies an offset from the beginning of the given segment to the desired byte. The offset must specify an address within the segment.

■ MAKEPGINFOSEG(*sel*)

This 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 previously retrieved by using the **DosGetInfoSeg** function.

Parameter	Description
<i>sel</i>	Specifies the segment selector of the global information segment.

■ MAKEPLINFOSEG(*sel*)

This 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 previously retrieved by using the **DosGetInfoSeg** function.

Parameter	Description
<i>sel</i>	Specifies the segment selector of the local information segment.

■ MAKESHORT(*l, h*)

This macro combines two 8-bit values to create a 16-bit integer.

Parameter	Description
<i>l</i>	Specifies the low-order 8-bit value of the new integer.
<i>h</i>	Specifies the high-order 8-bit value of the new integer.

■ MAKETYPE(*v, type*)

This macro casts the variable specified by the *v* parameter as a variable having the type specified by the *type* parameter. The macro permits the contents of the variable to be accessed as if the variable had the specified type.

Parameter	Description
<i>v</i>	Specifies the name of the variable to be cast.
<i>type</i>	Specifies the name of the data type for the cast.

■ MAKEULONG(*l, h*)

This macro combines two 16-bit values to create a 32-bit unsigned integer.

Parameter	Description
<i>l</i>	Specifies the low-order 16-bit value of the new integer.
<i>h</i>	Specifies the high-order 16-bit value of the new integer.

■ MAKEUSHORT(*l*, *h*)

This macro combines two 8-bit values to create a 16-bit unsigned integer.

Parameter	Description
<i>l</i>	Specifies the low-order 8-bit value of the new integer.
<i>h</i>	Specifies the high-order 8-bit value of the new integer.

■ OFFSETOF(*p*)

This macro retrieves the address offset of the specified far pointer.

Parameter	Description
<i>p</i>	Specifies a far (32-bit) pointer.

■ SELECTOROF(*p*)

This macro retrieves the selector from the specified far pointer.

Parameter	Description
<i>p</i>	Specifies a far (32-bit) pointer.

Appendix B

Devices

B.1	Introduction	527
B.2	Screen Modes	527
B.3	Screen Attributes	528
B.4	Physical Screen Buffer Addresses	529

B.1 Introduction

This appendix 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, and physical screen buffer addresses for the following display adapters:

IBM Monochrome/Printer Adapter
IBM Color Graphics Adapter (CGA)
IBM Enhanced Graphics Adapter (EGA)

These device-dependent values also apply to display adapters that are 100% compatible with the adapters listed above. MS OS/2 may also support display adapters not listed here.

B.2 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 pixels. 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 B.1, B.2, and B.3 list the screen modes available for the IBM Monochrome/Printer Adapter, Color Graphics Adapter, and Enhanced Graphics Adapter, and any adapter that is 100% compatible:

Table B.1

Screen Modes for Monochrome/Printer Adapter

Type	Colors	Row	Column	Horizontal	Vertical
Text	-	25	80	720	350

Table B.2**Screen Modes for CGA**

Type	Colors	Row	Column	Horizontal	Vertical
Text	16	25	40	320	200
Text	16	25	80	640	200
Graphics	4	25	40	320	200
Graphics	16	25	40	320	200
Graphics	2	25	80	640	200

Table B.3**Screen Modes for EGA**

Type	Colors	Row	Column	Horizontal	Vertical
Text	16	25	40	320	200
Text	16	25	40	320	350
Text	16	25	80	640	200
Text	16	25	80	640	350
Graphics	4	25	40	320	200
Graphics	16	25	40	320	200
Graphics	2	25	80	640	200
Graphics	4	25	80	640	200
Graphics	2	25	80	640	350
Graphics	16	25	80	640	200
Graphics	16	25	80	640	350

B.3 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 depend 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

B.4 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.

Appendix C

Code Pages

C.1	Introduction	533
C.2	Predefined Translation Tables	533
C.3	Translation-Table Format	533
C.4	Key Types	537
C.4.1	Alphabetic Key, Type 0x0001	539
C.4.2	Special Character Key, Type 0x0002	540
C.4.3	Special Character Key, Type 0x0003	541
C.4.4	Special Character Key, Type 0x0004	542
C.4.5	Special Character Key, Type 0x0005	543
C.4.6	Function Key, Type 0x0006	544
C.4.7	Keypad Key, Type 0x0007	545
C.4.8	Special Action Key, Type 0x0008	547
C.4.9	Print Screen Key, Type 0x0009	548
C.4.10	System Request Key, Type 0x000A	548
C.4.11	Accent Key, Type 0x000B	548
C.4.12	Shift Key, Type 0x000C	549
C.4.13	General Toggle Key, Type 0x000D	549
C.4.14	ALT Key, Type 0x000E	550
C.4.15	NUMLOCK Key, Type 0x000F	550
C.4.16	CAPSLOCK Key, Type 0x0010	551
C.4.17	SCROLL LOCK Key, Type 0x0011	551
C.4.18	Extended Shift Key, Type 0x0012	551
C.4.19	Extended Toggle Key, Type 0x0013	552
C.4.20	Special Foreign Key, Type 0x0014	553
C.4.21	Special Foreign Key, Type 0x0015	553

C.1 Introduction

This appendix 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.

C.2 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 identifier 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 identifier for the current system translation table.

The following is a list of the predefined translation tables and their code-page identifiers:

Code-Page ID	Meaning
437	IBM PC US
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. Also, the **keyb** commands can be used to change the current translation table.

C.3 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 identifier, 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 combinations keys. The table contains seven accent entries and accent-key definitions.

The translation table has the following form:

```
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 identifier for this translation.																		
XTableFlags1	Specifies the first set of table flags. It can be any combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0001</td><td>SHIFT+ALT key combination to be used in place of CONTROL+ALT.</td></tr><tr><td>0x0002</td><td>Left ALT key to be the alternate graphics key.</td></tr><tr><td>0x0004</td><td>Right ALT key to be the alternate graphics key.</td></tr><tr><td>0x0008</td><td>CAPSLOCK key to be interpreted as a SHIFT-LOCK key.</td></tr><tr><td>0x0010</td><td>Default table for the language. Used by the keyb command to locate the default translation table if switching between several translation tables.</td></tr><tr><td>0x0020</td><td>SHIFTLOCK key is a toggle key. If not given, the key is a latch key.</td></tr><tr><td>0x0040</td><td>Accent sent as character. If not valid, beeps.</td></tr><tr><td>0x0080</td><td>CAPS-SHIFT key uses the Char5 field in the key-definition entry.</td></tr></table>	Value	Meaning	0x0001	SHIFT+ALT key combination to be used in place of CONTROL+ALT.	0x0002	Left ALT key to be the alternate graphics key.	0x0004	Right ALT key to be the alternate graphics key.	0x0008	CAPSLOCK key to be interpreted as a SHIFT-LOCK 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 sent as character. If not valid, beeps.	0x0080	CAPS-SHIFT key uses the Char5 field in the key-definition entry.
Value	Meaning																		
0x0001	SHIFT+ALT key combination to be used in place of CONTROL+ALT.																		
0x0002	Left ALT key to be the alternate graphics key.																		
0x0004	Right ALT key to be the alternate graphics key.																		
0x0008	CAPSLOCK key to be interpreted as a SHIFT-LOCK 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 sent as character. If not valid, beeps.																		
0x0080	CAPS-SHIFT key uses the Char5 field in the key-definition entry.																		
XTableFlags2	Specifies a reserved value. It must be zero.																		
KbdType	Specifies the keyboard type. This field is 0x0000 for an IBM PC/AT keyboard. It is 0x0001 for an IBM Enhanced keyboard.																		
KbdSubType	Specifies a reserved value. It must be zero.																		
XTableLen	Specifies the length in bytes of the translation table.																		
EntryCount	Specifies the number of key-definition entries.																		
EntryWidth	Specifies the width in bytes of each key-definition entry.																		
Country	Specifies the country code or language identifier. The identifier 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. It can be any one of the following 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	Swiss-French
	SG	Swiss-German
	CF	Canadian-French
	BE	Belgium
	LA	Latin-American Spanish
TableTypeID	Specifies the table type. The low-order byte specifies the type, the high-order byte the subtype. 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 alternate graphics key is pressed.	
Char4	Specifies a translated character value.	

Char5	Specifies a translated character value.
NonAccent[2]	Specifies the character value and scan code for the key when not used as 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, with the first byte containing the scan code of a key to be accented and the second containing the character value of the accented key.

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 as described, a “not-an-accent” beep sounds whenever the accent key, followed by a space, is pressed.

C.4 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. The field can be one of the following values:

Value	Meaning
0x0001	Specifies an alphabetic-character key.
0x0002	Specifies a special nonalphabetic-character key.
0x0003	Special nonalphabetic-character key with CAPSLOCK translation.
0x0004	Special nonalphabetic-character key with ALT translation.
0x0005	Specifies a special nonalphabetic-character key with CAPSLOCK and ALT translations.
0x0006	Specifies a function key.

0x0007	Specifies a keypad key.
0x0008	Specifies an action key that performs a special action when the CONTROL key is pressed.
0x0009	Specifies the PRINTSCREEN key.
0x000A	Specifies the SYSREQ key.
0x000B	Specifies an accent key (also called a dead key).
0x000C	Specifies the SHIFT or CONTROL key.
0x000D	Specifies a general toggle key.
0x000E	Specifies the ALT key.
0x000F	Specifies the NUMLOCK key.
0x0010	Specifies the CAPSLOCK key.
0x0011	Specifies a SCROLL LOCK key.
0x0012	Specifies an extended shift key.
0x0013	Specifies an extended toggle key.
0x0014	Specifies a special character key with CAPSLOCK translations for foreign language keyboards.
0x0015	Specifies a 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 the key pressed immediately before the current key was an accent key, and if the bit for that accent key is set in the **AccentFlags** field for the current key, then the corresponding accent table entry is searched for the replacement character value to use. If no replacement is found, the “not-an-accent” beep sounds and the accent character and current character are passed as two separate characters. For more information, see Section C.4.11.

The SPACEBAR key 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 shifts are active, the **Char1** field specifies the translated character value (except where otherwise noted).

The ALT key or ALT-GRAPHICS 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-GRAPHICS key is specified, the **Char3** field specifies the translated-character value when the given key is pressed with the ALT-GRAPHICS key.

Any key combination that does not have an explicit definition is assumed to be undefined; for example, pressing the CONTROL key with 3. 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 C.1 to set and clear the keyboard shift-status word:

Table C.1
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	—

Sections C.4.1 through C.4.21 describe the key types in detail.

C.4.1 Alphabetic Key, Type 0x0001

An alphabetic key 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-GRAPHICS	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 is from 1 to 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.

C.4.2 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-GRAPHICS	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 a control code. A hyphen character (–) key always generates control code 0x1F, even if the corresponding scan code is not in the list. A hyphen character key is any key whose **Char1** field is 0x2D.

C.4.3 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.
ALT	Undefined translation.
ALT-GRAPHICS	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 a control code. A hyphen character (–) key always generates control code 0x1F, even if the corresponding scan code is not in the list. A hyphen character key is any key whose **Char1** field is 0x2D.

C.4.4 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-GRAPHICS	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 a control code. A hyphen character (–) key always generates control code 0x1F, even if the corresponding scan code is not in the list. 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.

C.4.5 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.
ALT	Computed extended ASCII code.
ALT-GRAPHICS	Char3 if this field is not zero.

Only the scan codes listed generate a control code. A hyphen character (-) key always generates control code 0x1F, even if the corresponding scan code is not in the list. 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.

C.4.6 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, that is, 1 for the F1 key, 2 for the F2 key, and so

on. The system generates the extended ASCII code in 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.
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	Add 0x67 to Char1 . The ALT+F11 and ALT+F12 keys are always 0x91 and 0x92, respectively.
ALT-GRAPHICS	Char3 if this field is not zero.

C.4.7 Keypad Key, Type 0x0007

A keypad key 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 (Note 6).
ALT-GRAPHICS	Char3 if this field is not zero.

The following list shows the required **Char1** values based on the keytop labels:

Keytop Label	Char1 Value
HOME/7	0x00
UP/8	0x01
PGUP/9	0x02
—	0x03
LEFT/4	0x04
5	0x05
RIGHT/6	0x06
+	0x07
END/1	0x08
DOWN/2	0x09
PGDN/3	0x0A
INS/0	0x0B
DEL/.	0x0C

The **Char2** values should represent the ASCII equivalent of the keytop 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 that is 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, the system builds a character value by accumulating keystrokes. On each keystroke, the system multiplies the accumulated value by 10 and 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 10 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:

Keytop Label	Extended Code
HOME/7	0x77
UP/8	0x8D
PGUP/9	0x84
—	0x8E
LEFT/4	0x73
5	0x8F
RIGHT/6	0x74
+	0x90
END/1	0x75
DOWN/2	0x91
PGDN/3	0x76
INS/0	0x92
DEL/.	0x93

C.4.8 Special Action Key, Type 0x0008

A special action key 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-GRAPHS	Char3 if this field is not zero.

C.4.9 Print Screen Key, Type 0x0009

The print screen key, PRINTSCREEN, 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-GRAPHICS	Char3 if this field is not zero.

C.4.10 System Request Key, Type 0x000A

The system request key represents a special shift key. The **Char1** field holds a bitmask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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.

C.4.11 Accent Key, Type 0x000B

An accent key (also called a dead key) represents a character to be 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 to 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-GRAPHICS	Char3

When an accent key is pressed with a CONTROL or ALT key, the system retrieves the character value from the **CtlAccent** or **AltAccent** 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, with a SHIFT key, or with the alternate graphics key, the system uses either the **Char1** or **Char2** 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 **XlateOp** field or the next key is not found in the **AccentMap** 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 alternate graphics 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.

C.4.12 Shift Key, Type 0x000C

A shift key 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 bitmask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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.

C.4.13 General Toggle Key, Type 0x000D

A general toggle key 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 bitmask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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.

C.4.14 ALT Key, Type 0x000E

The ALT key 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 bitmask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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 alternate graphics key, the ALT may be treated as that key.

C.4.15 NUMLOCK Key, Type 0x000F

The NUMLOCK key 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 bitmask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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.

C.4.16 CAPSLOCK Key, Type 0x0010

The CAPSLOCK key 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. A SHIFT key must be pressed to do so.

The **Char1** field holds a bitmask that the system uses to set the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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.

C.4.17 SCROLL LOCK Key, Type 0x0011

The SCROLL LOCK key 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 bitmask that the system uses to set or clear the lower byte of the keyboard shift-status word. The **Char2** field contains a bitmask 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.

C.4.18 Extended Shift Key, Type 0x0012

An extended shift key 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 bitmask 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 bitmask used to set or clear bits in the extended-status byte when the key is pressed or released.
Char3	Specifies the replacement bitmask 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**.

C.4.19 Extended Toggle Key, Type 0x0013

An extended toggle key represents a shift key that is used in conjunction with national language support. The key is similar to the toggle key, type 0x000D, 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 bitmask 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 bitmask used to set or clear bits in the extended status byte when the key is pressed.
Char3	Specifies the replacement bitmask 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**.

C.4.20 Special Foreign Key, Type 0x0014

A special foreign key represents any character that may need a CAPSLOCK translation:

Shift Key	Field Used
None	Char1
SHIFT	Char2
CAPSLOCK	Char4
CAPSLOCK and SHIFT	Char5
CONTROL	Computed ASCII control value.
ALT	No effect.
ALT-GRAPHS	Char3

C.4.21 Special Foreign Key, Type 0x0015

A special foreign key 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-GRAPHS	Char3

When ALT or ALT+SHIFT is pressed, the scan code and translated character code are equal.

Appendix D

Format of MS OS/2 Executable Files

D.1	Introduction	557
D.2	MS-DOS Executable Header	558
D.3	New Executable Header	558
D.4	Segment Table	561
D.5	Resource Table	562
D.6	Module-Reference Table	564
D.7	Entry-Point Table	564
D.8	Resident- and Nonresident-Name Tables	565
D.9	Imported-Names Table	566
D.10	Segments	566

D.1 Introduction

This appendix describes the format of the MS OS/2 executable files. This file format is used for both programs and dynamic-link libraries. The linker creates the executable file by using object files, run-time and import libraries, and a module-definition file. Figure D.1 shows the executable file format:

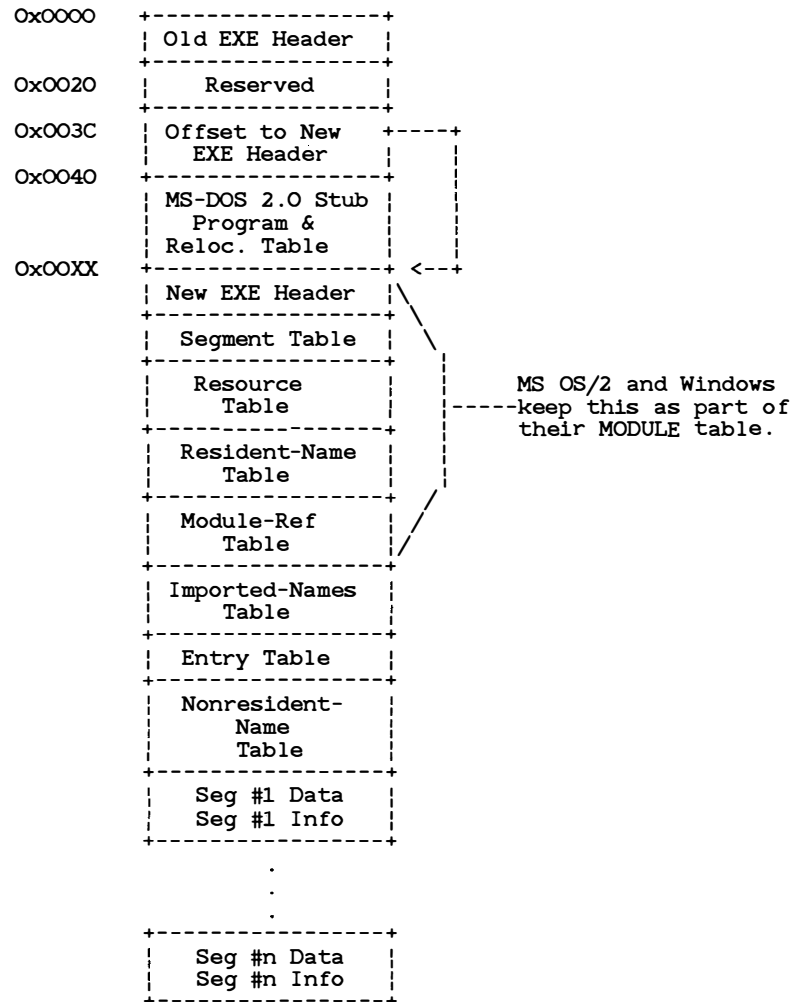


Figure D.1 Executable-File Format

D.2 MS-DOS Executable Header

An MS OS/2 executable file starts with a slightly modified MS-DOS executable header followed by the MS-DOS program data. The MS OS/2 executable file is a concatenation of an MS-DOS program and an MS OS/2 program. The MS-DOS program is usually a small stub program that displays an error message if a user loads and runs the executable file in MS-DOS. However, the linker can substitute a complete MS-DOS program for the stub.

In an MS-DOS executable file, the word at offset 0x0018 contains the relative byte offset to the relocation table. In an MS OS/2 executable file, this word is set to 0x0040 to indicate that the format of the executable file is for MS OS/2. In such cases, the double word at offset 0x003C is the relative byte offset from the beginning of the MS OS/2 executable file to the beginning of the executable file header.

The following list shows the contents of the various fields in the MS-DOS program portion of the MS OS/2 executable file:

Field	Description
0x0000–0x001F	Contains the header for the MS-DOS executable file.
0x0018	Contains the value 0x0040 to indicate an MS OS/2 executable file.
0x0020–0x003B	Contains reserved values.
0x003C–0x003F	Contains a double word that specifies the offset from the beginning of the MS OS/2 file to the start of the MS OS/2 executable file header.
0x0040	Contains the first byte of the MS-DOS program. The length is defined in the MS-DOS executable file header.

D.3 New Executable Header

The new executable header defines the location and size of the various tables and segments in the executable file. The MS OS/2 loader uses this header to create a module table for each program and each dynamic-link library. Specifically, it uses the fields that start at offset 0x0008.

Many fields in the executable header use segment numbers to identify segments in the program or library. A segment number is an index into the module's segment table. The first entry in the segment table is number 1.

In the following list, 0x0000 is specified as the starting offset of the executable file. The actual starting offset depends on the length of the MS-DOS program at the beginning of the executable file. The executable file header always starts immediately after the end of the MS-DOS program.

Field	Description																		
0x0000	Specifies the signature of the MS OS/2 executable file. It is 0x454E (that is, the letters “NE” for “new executable”).																		
0x0002	Contains the linker version and the revision number. The linker version is in the low-order byte, the revision number is in the high-order byte.																		
0x0004	Specifies the offset from the beginning of the file to the entry table.																		
0x0006	Specifies the number of bytes in the entry table.																		
0x0008–0x000A	Specifies the CRC-32 of the entire contents of the file (with the following words taken as zero during the calculation).																		
0x000C	Specifies the flag word. It can be a combination of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>No automatic data segment.</td></tr><tr><td>0x0001</td><td>Single data segment (nonshared).</td></tr><tr><td>0x0002</td><td>Multiple data segments (shared).</td></tr><tr><td>0x0004</td><td>Run in real mode.</td></tr><tr><td>0x0008</td><td>Run in protected mode.</td></tr><tr><td>0x2000</td><td>Errors detected at link time.</td></tr><tr><td>0x4000</td><td>Non-conforming program (a valid stack is not maintained).</td></tr><tr><td>0x8000</td><td>Library module.</td></tr></table> The 0x0001 and 0x0002 values cannot be used together. The 0x0001 value is required for programs or libraries that export functions for dynamic linking.	Value	Meaning	0x0000	No automatic data segment.	0x0001	Single data segment (nonshared).	0x0002	Multiple data segments (shared).	0x0004	Run in real mode.	0x0008	Run in protected mode.	0x2000	Errors detected at link time.	0x4000	Non-conforming program (a valid stack is not maintained).	0x8000	Library module.
Value	Meaning																		
0x0000	No automatic data segment.																		
0x0001	Single data segment (nonshared).																		
0x0002	Multiple data segments (shared).																		
0x0004	Run in real mode.																		
0x0008	Run in protected mode.																		
0x2000	Errors detected at link time.																		
0x4000	Non-conforming program (a valid stack is not maintained).																		
0x8000	Library module.																		
0x000E	Specifies the segment number of the automatic data segment. This field is set to zero if the module has no automatic data segment.																		
0x0010	Specifies the initial size in bytes of the dynamic heap added to the data segment. This field is set to zero if there is no heap.																		

0x0012	Specifies the initial size in bytes of the stack added to the data segment. This field is set to zero if there is no stack.
0x0014–0x0016	Specifies the starting address of the program or of the library's initialization function. The low-order word is the offset (IP); the high-order word is the segment number of the starting segment (CS). The address specifies the entry point of the initialization function only if the executable file defines a library module.
0x0018–0x001A	Specifies the starting address of the stack. The low-order word is the offset (SP); the high-order word is the segment number of the stack segment (SS). If the number of the stack segment is the same as the number of the automatic data segment and the offset is zero, the loader creates a stack in the automatic data segment and sets the starting address to be the address of the last word in that stack. This field is ignored if the executable file defines a library module.
0x001C	Specifies the number of entries in the segment table.
0x001E	Specifies the number of bytes in the nonresident-name table.
0x0020	Specifies the offset of the segment table relative to the beginning of the new executable header.
0x0022	Specifies the offset of the resource table relative to the beginning of the new executable header.
0x0024	Specifies the offset of the resident-name table relative to the beginning of the new executable header.
0x0026	Specifies the offset of the module-reference table relative to the beginning of the new executable header.
0x0028	Specifies the offset of the imported-names table relative to the beginning of the new executable header.
0x002A–0x002C	Specifies the offset of the nonresident-name table relative to the beginning of the file.
0x002E	Specifies the number of movable entry points.
0x0030	Specifies the shift count of the logical sector alignment. The alignment specifies the byte boundary on which a segment starts. It is expressed as an exponent of 2; for the default alignment of 512 bytes, the shift count 9 is given.
0x0032–0x003E	Specifies reserved values.

The system loader creates a stack for a program if the stack segment is the same as the automatic data segment. The loader adds the stack to the end of the data segment, setting the stack pointer to the top of the automatic data segment. If the data segment also has a heap area, the stack is between the data and heap.

D.4 Segment Table

The segment table defines the location, size, and type of the code and data segments of the module. The segment table contains one or more entries. Each entry specifies the location of the code and data segments in the executable file, the size of the segment data in the file, the size of the segment when loaded into memory, and a flag word that specifies the segment type. The number of segment entries in the table is specified by field 0x001C in the executable file header.

Many fields in the executable file header use segment numbers to identify segments in this table. Segment number 1 identifies the first segment table entry, number 2 the second, and so on.

Each entry in the segment table has the following fields:

Field	Description												
0x0000	Specifies the logical sector offset to the contents of the segment data relative to the beginning of the file. The actual offset is computed by multiplying this offset with the logical sector size as defined by field 0x0030 in the executable file header. This field is zero if there is no file data.												
0x0002	Specifies the length in bytes of the segment in the file. It is zero if the segment is 65,536 bytes.												
0x0004	Specifies the flag word. It can be a combination of the following values:												
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0000</td><td>Code-segment type.</td></tr><tr><td>0x0001</td><td>Data-segment type.</td></tr><tr><td>0x0008</td><td>Segment data is iterated.</td></tr><tr><td>0x0010</td><td>Segment is movable.</td></tr><tr><td>0x0020</td><td>Segment can be shared.</td></tr></table>	Value	Meaning	0x0000	Code-segment type.	0x0001	Data-segment type.	0x0008	Segment data is iterated.	0x0010	Segment is movable.	0x0020	Segment can be shared.
Value	Meaning												
0x0000	Code-segment type.												
0x0001	Data-segment type.												
0x0008	Segment data is iterated.												
0x0010	Segment is movable.												
0x0020	Segment can be shared.												

	0x0040	Segment is not demand loaded.
	0x0080	Segment is execute-only if code, or read-only if data.
	0x0100	Set if segment has relocation records.
	0x0200	Set if segment has debug information.
	0x0C00	Reserved for 286 DPL bits.
	0xF000	Discard priority.
0x0006		Specifies the minimum allocation size of the segment in bytes. This may be larger than the size of the segment in the file. If this field is zero, the minimum size is 65,536 bytes.

D.5 Resource Table

The resource table specifies the location, size, type, and name of resources in the module. Resources are additional data that may be loaded from the executable file as needed by a program or library. Figure D.2 shows the form of the resource table:

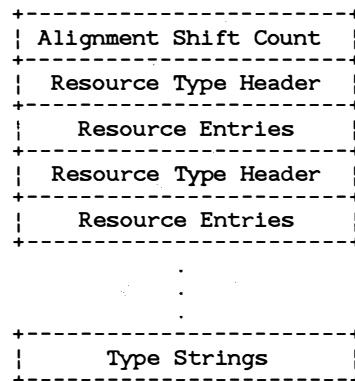


Figure D.2 Resource Table

The first word in the resource table specifies the alignment shift count for resource data. The alignment shift count is an exponent of 2 that defines the number of bytes in each alignment sector. The location of a resource in the executable file is computed by multiplying its alignment offset by the alignment sector size.

The resource table contains two or more resource-type headers. Each header has the following fields:

Field	Description
0x0000	Specifies the type identifier. It is an integer type if the high-order bit is set (0x8000). Otherwise, it is an offset to a type string, relative to the beginning of the resource table. If this field is zero, it specifies the end of the resource records.
0x0002	Specifies the number of resources for the type.
0x0004–0x0006	Specifies a reserved value.

The resource-type header is immediately followed by the specified number of resource entries. Each resource entry has the following fields:

Field	Description								
0x0000	Specifies the alignment offset to the contents of the resource data, relative to the beginning of the file. The offset is given in terms of alignment units specified at the beginning of the resource table.								
0x0002	Specifies the length in bytes of the resource in the file.								
0x0004	Specifies the flag word. It can be a combination of the following values:								
	<table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x0010</td><td>Resource is movable.</td></tr><tr><td>0x0020</td><td>Resource can be shared.</td></tr><tr><td>0x0040</td><td>Resource is not demand loaded.</td></tr></table>	Value	Meaning	0x0010	Resource is movable.	0x0020	Resource can be shared.	0x0040	Resource is not demand loaded.
Value	Meaning								
0x0010	Resource is movable.								
0x0020	Resource can be shared.								
0x0040	Resource is not demand loaded.								
0x0006	Specifies the resource identifier. It is an integer type if the high-order bit is set (0x8000). Otherwise, this field is the offset to the resource string relative to the beginning of the resource table.								
0x0008–0x000A	Specifies a reserved value.								

Resource type and name strings are stored at the end of the resource table. The first byte of each string specifies the length of the string in bytes. If it is zero, it specifies the end of the resource table. The strings can contain any characters. They are not null-terminated.

D.6 Module-Reference Table

The module-reference table specifies the location of the names of the modules imported by the module. The names are character strings and are stored in the imported-names table.

The table contains one or more entries. Each entry specifies the offset (within the imported-names table) to the module-name string. Each entry has a unique module reference. A module reference is an index into the module-reference table. Module reference 1 identifies the first entry, 2 the second, and so on. Other tables use module references to identify the name of the module that contains a given imported function.

D.7 Entry-Point Table

The entry-point table defines the segment, offset, and type of entry points used in the module. The table contains one or more bundles. Each bundle describes the entry points of a given segment. Each bundle starts with the following fields:

Field	Description
0x0000	Specifies the number of entries in this bundle. All records in one bundle either are movable or refer to the same fixed segment. This field is zero if there are no more bundles in the entry table.
0x0001	Specifies the segment indicator for this bundle. If it is 0xFF, it specifies a movable segment, and the segment number is in the following 6-byte entry. If it is any other value (0x00 is not used), it is the segment number of a fixed segment.

For movable segments, the entry record has the following fields:

Field	Description
0x0000	Specifies the flags. If it is 0x01, the entry point is exported. If it is 0x02, the entry point uses a shared data segment.
0x0001	Specifies an int 3Fh instruction.
0x0003	Specifies the number of the entry-point segment.
0x0004	Specifies the offset to the entry point.

For fixed segments, the entry record has the following fields:

Field	Description
0x0000	Specifies the flags. If it is 0x01, the entry point is exported. If it is 0x02, the entry point uses a shared data segment.
0x0001	Specifies the offset to the entry point.

For movable and fixed entry points that use shared data segments, the following instruction must be the first instruction in the prologue of the entry point:

mov ax, ds-value

This flag may be set only for SINGLEDATA library modules.

Each entry point has a unique ordinal value. The ordinal value is an index into the entry-point table. The first entry point described in the table has ordinal value 1, the second has 2, and so on. When the loader searches for an entry point, it scans the bundles until it finds the segment that contains the entry point. It then multiplies the ordinal value by the entry size to index the proper entry.

The linker forms bundles in the densest manner it can, under the restriction that it cannot reorder entry points to improve bundling. The reason for this restriction is that other executable files may refer to entry points within this bundle by ordinal value instead of by name.

D.8 Resident- and Nonresident-Name Tables

The resident- and nonresident-name tables specify the names of the entry points in the module. The tables contain one or more entries. Each entry has the form shown in Figure D.3:

Length of Name
Name
Ordinal Value

Figure D.3 Resident- and Nonresident-Name Table Entry Format

The first field is a byte that specifies the length of the string. If the field is zero, it specifies the end of the table. The second field is the name, a character string. The string is not null-terminated. The last field is the entry-point ordinal value.

The first string in the resident-name table is the module name. The first string in the nonresident-name table is the module description.

D.9 Imported-Names Table

The imported-names table contains the names of modules that contain entry points imported by the module. The table contains one or more entries. Each entry starts with a byte that specifies the length of the name in bytes. If the byte is zero, it specifies the end of the table. The byte is followed immediately by the name, a character string. The string is not null-terminated.

D.10 Segments

The segments contain the data that is to be loaded into memory for the program or library. The format of the data depends on the segment type, as specified by field 0x0004 in the segment table.

If the segment is not iterated, it consists of the number of bytes specified in field 0x0002 in the segment table. If the segment has iterated data, the segment has the following fields:

Field	Description
0x0000	Specifies the number of iterations.
0x0002	Specifies the number of bytes of data.
0x0004	Specifies the data bytes to be repeated.

If the segment has relocation information, the relocation information consists of one or more relocation items. The relocation information starts with a word that specifies the number of relocation items. Each relocation item has the following fields:

Field	Description										
0x0000	Specifies the relocation type. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x02</td><td>16-bit segment selector or address.</td></tr><tr><td>0x03</td><td>32-bit address.</td></tr><tr><td>0x05</td><td>16-bit address offset.</td></tr></table>	Value	Meaning	0x02	16-bit segment selector or address.	0x03	32-bit address.	0x05	16-bit address offset.		
Value	Meaning										
0x02	16-bit segment selector or address.										
0x03	32-bit address.										
0x05	16-bit address offset.										
0x0001	Specifies the relocation flags. It can be one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>0x00</td><td>Internal reference.</td></tr><tr><td>0x01</td><td>Imported by ordinal.</td></tr><tr><td>0x02</td><td>Imported by name.</td></tr><tr><td>0x04</td><td>Additive reference.</td></tr></table>	Value	Meaning	0x00	Internal reference.	0x01	Imported by ordinal.	0x02	Imported by name.	0x04	Additive reference.
Value	Meaning										
0x00	Internal reference.										
0x01	Imported by ordinal.										
0x02	Imported by name.										
0x04	Additive reference.										
0x0002	Specifies the offset within the segment of the source chain. If the additive flag is set, then add the target value to the source contents instead of replacing the source and following the chain. The source chain is a linked list within this segment of all references to the target. The list ends with the value 0xFFFF.										

The source for each relocation has a specific format based on the relocation flags. The following lists define each of the source formats.

An INTERNALREF source has the following fields:

Field	Description
0x0000	Specifies the segment number for the fixed segment, or 0xFF if the segment is movable.
0x0001	Specifies a reserved value. It must be zero.
0x0002	Specifies the ordinal value of the entry point if the segment is movable. Otherwise, it specifies the offset into the segment.

An IMPORTNAME source has the following fields:

Field	Description
0x0000	Specifies the module reference. For more information, see Section D.6.
0x0002	Specifies the offset within the imported-names table to the function name.

An IMPORTORDINAL source has the following fields:

Field	Description
0x0000	Specifies the module reference. For more information, see Section D.6.
0x0002	Specifies the ordinal value for the imported function.

If a segment has debugging information, the first word specifies the number of bytes of debugging information. The remaining bytes are the actual debugging information.

Appendix E

ANSI Escape Sequences

E.1	Introduction	571
E.2	Cursor Functions	571
E.2.1	Cursor Position	571
E.2.2	Cursor Up	572
E.2.3	Cursor Down	572
E.2.4	Cursor Forward	572
E.2.5	Cursor Backward	572
E.2.6	Save Cursor Position	572
E.2.7	Restore Cursor Position	572
E.3	Erase Functions	573
E.3.1	Erase Display	573
E.3.2	Erase Line	573
E.4	Screen Graphics Functions	573
E.4.1	Set Graphics Rendition	573
E.4.2	Set Mode	574
E.4.3	Reset Mode	575

10

11

12

E.1 Introduction

This appendix lists all of the escape sequences that can be used with the *ansi.sys* installable device driver in real mode, or with the **ansi** command in protected mode.

Escape sequences used in the *ansi.sys* file affect cursor positioning, erase functions, and screen graphics.

The ANSI escape sequences listed in this appendix use the following variables:

Variable	Description
<i>n</i>	A number, as in the number of rows or columns a cursor moves
<i>col</i>	A column number
<i>row</i>	A row number

The sequences must be typed exactly as shown. No spaces are allowed.

E.2 Cursor Functions

The following functions affect the positioning of the cursor.

E.2.1 Cursor Position

ESC [*row* ; *col* H

or

ESC [*row* ; *col* f

These two escape 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).

E.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.

E.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.

E.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.

E.2.5 Cursor Backward

ESC [*n* D

This escape 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.

E.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.

E.2.7 Restore Cursor Position

ESC [u

This sequence restores the cursor position to the Save Cursor Position value.

E.3 Erase Functions

The following functions erase the screen.

E.3.1 Erase Display

ESC [2 J

This sequence erases the screen and moves the cursor to the home position (the upper-left corner of the screen).

E.3.2 Erase Line

ESC [K

This sequence erases from the cursor to the end of the line (including the cursor position).

E.4 Screen Graphics Functions

The following functions affect screen graphics.

E.4.1 Set Graphics Rendition

ESC [*g*; ... ; *g* m

This escape sequence calls the graphics functions specified by the following numeric parameters. These functions remain until the next occurrence of this escape sequence. This escape 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.
5	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.

Parameters 30 through 47 meet the ISO 6429 standard.

E.4.2 Set Mode

ESC =_s h

This escape sequence changes the screen width or type. The *s* argument can be one of the following numeric parameters:

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. |

E.4.3 Reset Mode

ESC =s l

Parameters for this escape sequence are the same as for Set Mode except parameter 7 resets the mode that causes wrapping at the end of each line.

(

(

(

Index

- access shared segment, 132
- ALLOWPTRDRAW, 398
- ansi command
 - protected mode, 571
- ansi.sys device driver, 571
- ansi.sys file
 - escape sequences, 571
- asynchronous read from a file, 192
- asynchronous timer, 247
- asynchronous write to a file, 253

- BLOCKREMOVABLE, 398

- change file size, 155
- change size, 197, 198
- character input, keyboard, 270
- clear semaphore, 206
- code page, 265, 277, 278
- CONTROL+BREAK, 213
- CONTROL+C, 213
- current directory, 177
- cursor
 - blink, 573
 - erasing characters, 573
 - home position, 571
 - movement, 571
 - position, 571
 - positioning, 571
 - saving the current position, 572

- date, get, 113
- date, set, 214
- debugging, 171
- DEFINEMUXSEMLIST, 519
- device driver
 - ansi.sys, 571
- disk drive, select, 204
- disk partition, 168
- disk, query current, 179
- DOS version number, 133
- DosAllocHuge, 47
- DosAllocHuge shift count, get, 119
- DosAllocSeg, 49
- DosAllocShrSeg, 52
- DosBeep, 53
- DosBufReset, 54
- DosCaseMap, 55
- DosChdir, 57
- DosChgFilePtr, 58
- DosCLIAccess, 60
- DosClose, 61
- DosCloseQueue, 62
- DosCloseSem, 63
- DosCreateCSAlias, 64
- DosCreateQueue, 65
- DosCreateSem, 66
- DosCreateThread, 68
- DosCWait, 70
- DosDelete, 73
- DosDevConfig, 74
- DosDevIOctl, 75
- DosDupHandle, 77
- DosEnterCritSec, 79
- DosErrClass, 79
- DosError, 82
- DosExecPgm, 84
- DosExit, 90
- DosExitCritSec, 91
- DosExitList, 92
- DosFileLocks, 94
- DosFindClose, 96
- DosFindFirst, 98
- DosFindNext, 101
- DosFlagProcess, 104
- DosFreeModule, 106
- DosFreeSeg, 106
- DosGetCollate, 107
- DosGetCp, 109
- DosGetCtryInfo, 110
- DosGetDateTime, 113
- DosGetDBCSEv, 115
- DosGetEnv, 117
- DosGetHugeShift, 119
- DosGetInfoSeg, 119
- DosGetMachineMode, 123
- DosGetMessage, 124
- DosGetModHandle, 126
- DosGetModName, 127
- DosGetPid, 128
- DosGetProcAddr, 129
- DosGetPrty, 130
- DosGetSeg, 132
- DosGetShrSeg, 132
- DosGetVersion, 133
- DosGiveSeg, 134
- DosHoldSignal, 135
- DosInsMessage, 136

DosKillProcess, 138
 DosLoadModule, 139
 DosLockSeg, 141
 DosMakePipe, 142
 DosMemAvail, 143
 DosMkdir, 144
 DosMonClose, 144
 DosMonOpen, 145
 DosMonRead, 146
 DosMonReg, 147
 DosMonWrite, 149
 DosMove, 150
 DosMuxSemWait, 152
 DosNewSize, 155
 DosOpen, 156
 DosOpenQueue, 163
 DosOpenSem, 164
 DosPeekQueue, 165
 DosPhysicalDisk, 168
 DosPortAccess, 170
 DosPTrace, 171
 DosPurgeQueue, 176
 DosPutMessage, 176
 DosQCurDir, 177
 DosQCurDisk, 179
 DosQFHandState, 180
 DosQFileInfo, 182
 DosQFileMode, 184
 DosQFSInfo, 186
 DosQHandType, 188
 DosQueryQueue, 189
 DosQVerify, 190
 DosRead, 191
 DosReadAsync, 192
 DosReadQueue, 195
 DosReallocHuge, 197
 DosReallocSeg, 198
 DosResumeThread, 199
 DosRmdir, 200
 DosScanEnv, 201
 DosSearchPath, 202
 DosSelectDisk, 204
 DosSelectSession, 205
 DosSemClear, 206
 DosSemRequest, 207
 DosSemSet, 209
 DosSemSetWait, 210
 DosSemWait, 211
 DosSendSignal, 213
 DosSetCp, 214
 DosSetDateTime, 214
 DosSetFHandState, 216
 DosSetFileInfo, 218
 DosSetFileMode, 220
 DosSetFSInfo, 221
 DosSetMaxFH, 222

DosSetPrty, 223
 DosSetSession, 225
 DosSetSigHandler, 228
 DosSetVec, 231
 DosSetVerify, 233
 DosSleep, 234
 DosStartSession, 235
 DosStopSession, 237
 DosSubAlloc, 238
 DosSubFree, 239
 DosSubSet, 240
 DosSuspendThread, 241
 DosSystemService, 242
 DosTimerAsync, 247
 DosTimerStart, 248
 DosTimerStop, 250
 DosUnlockSeg, 250
 DosWrite, 251
 DosWriteAsync, 253
 DosWriteQueue, 255
 DRAWPTR, 399

environment string, 117
 erase function, 571
 .exe file format
 contents, 557

family API, 38
 FIELDOFFSET, 520
 file

 ansi.sys
 escape sequences, 571
 asynchronous, read from, 192
 asynchronous, write to, 253
 format, executable, 557
 I/O, 251
 open, 156
 query mode, 184
 read from, 191
 set attribute, 220
 set mode, 220
 size, change, 155
 synchronous write, 251
 write to a, 251
 file handle state, set, 216
 file handle state, query, 180
 file handles, set maximum, 222
 file information, query, 182
 file information, set, 218
 file system ID, query, 186
 file system ID, set, 221
 file system information, query, 186
 flag process, 104
 flush buffer, keystroke, 263

FLUSHINPUT, 399
FLUSHOUTPUT, 400
FORMATVERIFY, 401
free segment, 106

get DosAllocHuge shift count, 119
get priority, 130
get shared segment, 132
get shift count, 119
get version number, 133
GETBAUDRATE, 402
GETBUTTONCOUNT, 403
GETCOMMERROR, 403
GETCOMMEVENT, 404
GETCOMMSTATUS, 406
GETDCBINFO, 408
GETDEVICEPARAMS, 410
GETEVENTMASK, 413
GETFRAMECTL, 414
GETHOTKEYBUTTON, 415
GETINFINITERETRY, 416
GETINPUTMODE, 416
GETINQUECOUNT, 417
GETINTERIMFLAG, 418
GETKEYBDTYPE, 419
GETLINECTRL, 420
GETLINESTATUS, 422
GETLOGICALMAP, 422
GETMICKEYCOUNT, 423
GETMODEMINPUT, 424
GETMODEMOUTPUT, 424
GETMOUSTATUS, 425
GETOUTQUECOUNT, 426
GETPHYSDEVICEPARAMS, 427
GETPRINTERSTATUS, 428
GETPTRDRAWADDRESS, 429
GETPTRPOS, 430
GETPTRSHAPE, 431
GETQUESTATUS, 433
GETSCALEFACTORS, 434
GETSESMGRHOTKEY, 435
GETSHIFTSTATE, 437
give segment, 134

handle type, query, 188
HIBYTE, 520
HIUCHAR, 520
HIUSHORT, 520
hold signal, 135
huge reallocation, 197

INITPRINTER, 438
interval timer, 248, 250

KbdCharIn, 258
KbdClose, 261
KbdDeRegister, 262
KbdFlushBuffer, 263
KbdFreeFocus, 264
KbdGetCp, 265
KbdGetFocus, 266
KbdGetStatus, 267
KbdOpen, 269
KbdPeek, 270
KbdRegister, 273
KbdSetCp, 277
KbdSetCustXt, 278
KbdSetStatus, 279
KbdStringIn, 282
KbdSynch, 284
KbdXlate, 285
keyboard flush buffer, 263
keyboard input, 270

LOBYTE, 521
LOCKDRIVE, 439
LOCKPHYSDRIVE, 439
logical keyboard, 269
LOUCHAR, 521
LOUSHORT, 521

make subdirectory, 144
MAKEERRORID, 521
MAKELONG, 522
MAKEP, 522
MAKEPGINFOSEG, 522
MAKEPLINFOSEG, 523
MAKESHORT, 523
MAKETYPE, 523
MAKEULONG, 523
MAKEUSHORT, 524
message retriever, 124, 136
message text output, 176
monitor close, 144
monitor open, 145
monitor read, 146
monitor register, 147
monitor write, 149
MouClose, 288
MouDeRegister, 288
MouDrawPtr, 289
MouFlushQue, 290
MouGetDevStatus, 291
MouGetEventMask, 292
MouGetHotKey, 293
MouGetNumButtons, 295
MouGetNumMickey, 295
MouGetNumQueEl, 296

MouGetPtrPos, 298
 MouGetPtrShape, 299
 MouGetScaleFact, 301
 MouInitReal, 303
 MouOpen, 304
 MouReadEventQue, 305
 MouRegister, 307
 MouRemovePtr, 311
 MouSetDevStatus, 313
 MouSetEventMask, 315
 MouSetHotKey, 316
 MouSetPtrPos, 317
 MouSetPtrShape, 319
 MouSetScaleFact, 321
 MouSynch, 323
 multiple wait, 152

notational conventions, 9

OFFSETOF, 524
 open file, 156
 open queue, 163
 open semaphore, 164

pause, 234
 peek queue, 165
 PEEKCHAR, 440
 pipe, 142
 pipe, make, 142
 priority, 223
 priority, get, 130
 process, stop, 138
 program terminate, 138
 purge queue, 176

QUERYMONSUPPORT, 442

queue
 open, 163
 peek, 165
 purge, 176
 query, 189
 read, 195
 write, 255

read from a file, 191
 read, keyboard character, 270
 read, keyboard scan code, 270
 read queue, 195
 READCHAR, 443
 READEVENTQUE, 445
 READPHYSRACK, 447

READTRACK, 448
 reallocate huge memory, 197
 reallocate segment, 198
 REDETERMINEMEDIA, 450
 register keyboard subsystem, 273
 REGISTERMONITOR, 451
 release semaphore, 206
 remove subdirectory, 200
 REMOVEPTR, 452
 request semaphore, 207
 resume thread, 199
 reverse video, 573

scan code, 285

screen

 colors, 574
 erasing, 573
 graphics, 573, 574
 moving the cursor, 571
 type, 574
 width, 574

screen graphics

 ansi.sys device driver, 571

SCREENSWITCH, 453

segment, 132, 250

segment, give away, 134

select disk drive, 204

SELECTOROF, 524

semaphore

 clear, 206
 open, 164
 release, 206
 request, 207
 set, 209
 wait for, 210, 211
 wait for multiple, 152

session, 237

set a file's information, 218

set file attribute, 220

set file handle state, 216

set file mode, 220

set file system ID, 221

set file system information, 221

set interrupt vector, 231

set maximum file handle, 222

set priority, 223

set semaphore, 209

set signal handler, 228

set verify switch, 233

SETBAUDRATE, 454

SETBREAKOFF, 455

SETBREAKON, 456

SETDCBINFO, 457

SETDEVICEPARAMS, 475

SETEVENTMASK, 478

- SETFGNDSCREENGRP, 478
- SETFOCUS, 479
- SETFRAMECTL, 480
- SETHOTKEYBUTTON, 481
- SETINFINITERETRY, 482
- SETINPUTMODE, 482
- SETINTERIMFLAG, 483
- SETLINECTRL, 484
- SETLOGICALMAP, 486
- SETMODEMCTRL, 487
- SETMOUSTATUS, 489
- SETPROTDRAWADDRESS, 490
- SETPTRPOS, 491
- SETPTRSHAPE, 492
- SETREALDRAWADDRESS, 493
- SETSCALEFACTORS, 494
- SETSESMGRHOTKEY, 495
- SETSHIFTSTATE, 497
- SETTRANSTABLE, 499
- SETTYPAMATICRATE, 500
- shared, segment access, 132
- shift count, get, 119
- signal, 135
- signal handler, 228
- size, change, 197, 198
- sleep, delay, process, 234
- sleep process, 234
- STARTTRANSMIT, 501
- stop process, 138
- STOPTRANSMIT, 501
- subdirectory, make, 144
- subdirectory, remove, 200
- suspend thread, 241
- system task services, 242
- system variables, 119
- terminate program, 138
- thread, resume, 199
- thread, suspend, 241
- time, get, 113
- time set, 214
- timer, interval, 248, 250
- timer, start, asynchronous, 247
- translation table, 265, 277, 278, 285
- TRANSMITIMM, 502
- unlock segment, 250
- UNLOCKDRIVE, 504
- UNLOCKPHYSDRIVE, 504
- UPDATEDISPLAYMODE, 505
- verify setting, query, 190
- verify switch, set, 233
- VERIFYPHYSTRACK, 506
- VERIFYTRACK, 508
- VioDeRegister, 324
- VioEndPopUp, 324
- VioGetAnsi, 325
- VioGetBuf, 326
- VioGetConfig, 327
- VioGetCp, 329
- VioGetCurPos, 330
- VioGetCurType, 331
- VioGetFont, 333
- VioGetMode, 335
- VioGetPhysBuf, 337
- VioGetState, 339
- VioModeUndo, 341
- VioModeWait, 343
- VioPopUp, 344
- VioPrtSc, 347
- VioPrtScToggle, 348
- VioReadCellStr, 348
- VioReadCharStr, 350
- VioRegister, 351
- VioSavRedrawUndo, 355
- VioSavRedrawWait, 357
- VioScrLock, 359
- VioScrollDn, 360
- VioScrollLf, 362
- VioScrollRt, 364
- VioScrollUp, 365
- VioScrUnLock, 367
- VioSetAnsi, 368
- VioSetCp, 369
- VioSetCurPos, 370
- VioSetCurType, 371
- VioSetFont, 372
- VioSetMode, 374
- VioSetState, 376
- VioShowBuf, 379
- VioWrtCellStr, 380
- VioWrtCharStr, 381
- VioWrtCharStrAtt, 383
- VioWrtNAttr, 384
- VioWrtNCell, 385
- VioWrtNChar, 387
- VioWrtTTY, 388
- wait for semaphore, 210, 211
- write queue, 255
- write to a file, 251
- WRITEPHYSTRACK, 510
- WRITETRACK, 512
- yield time slice, 234

