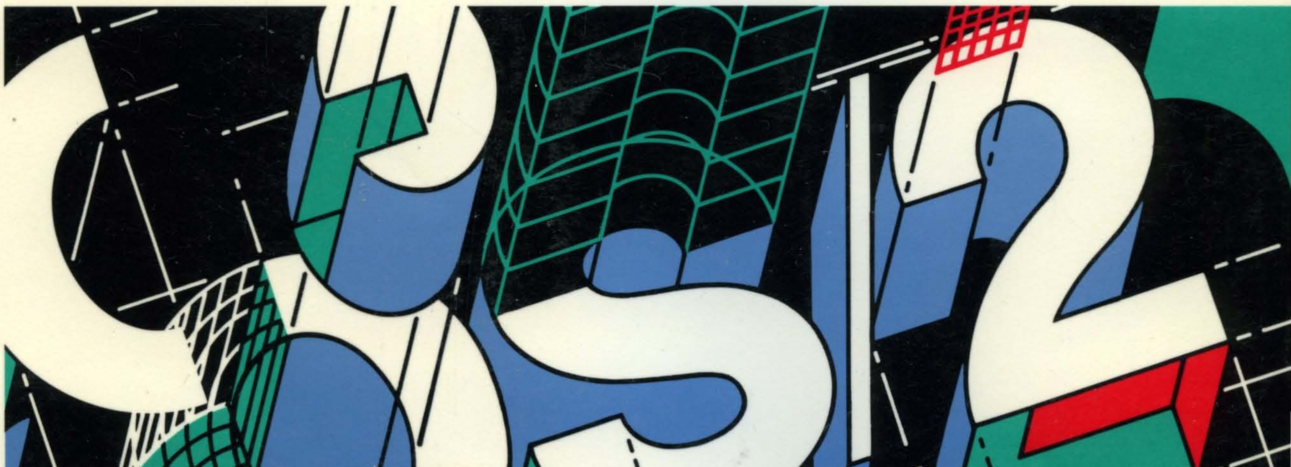
Volume 4

MICROSOFT®

OS/2 Programmer's Reference



Including Presentation Manager

Microsoft OS|2

PROGRAMMER'S REFERENCE LIBRARY

Volume 4

Microsoft® Operating System/2 Programmer's Reference

Version 1.2

Written, edited, and produced by Microsoft Corporation

Distributed by Microsoft Press

Microsoft OS|2

PROGRAMMER'S REFERENCE LIBRARY

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Cataloging-in-Publication Data
(Revised for vol. 4)
Microsoft OS/2 programmer's reference.
(Microsoft OS/2 programmer's reference library)
Vol. 4 has title: Microsoft Operating System/2 programmer's reference.
Includes indexes.
Contents: v. 1 – – –v. 4. Version 1.1/ written, edited, and produced by Microsoft Corporation.
1. OS/2 (Computer operating system). I. Microsoft Press. II. Microsoft III. Title: Microsoft Operating System/2 programmer's reference.
QA76.76.063M5 1989 005.4'469 89-2817
ISBN 1-55615-220-5 (v. 1)

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

Distributed to the book trade in Canada by General Publishing Company, Ltd.
Distributed to the book trade outside the United States and Canada by Penguin Books Ltd.
Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182–190 Wairau Road, Auckland 10, New Zealand
British Cataloging in Publication Data available

Patent #4,825,358
Patent #4,779,187

Contents

Chapter 1 Introduction

1.1 Overview 3
1.2 How to Use This Manual 4
1.3 Naming Conventions 8
1.4 Notational Conventions 11

Chapter 2 Overviews

2.1 Introduction 17
2.2 Installable File Systems 17
2.3 Extended Attributes 23
2.4 Profile Manager 28
2.5 Help Manager 32
2.6 Combination-Box Controls 40
2.7 Multiple-Line Entry Fields 43

Chapter 3 Functions and Messages Directory

3.1 Introduction 53
3.2 Directory 55
3.3 Functions and Messages 56

Chapter 4 Types, Macros, Structures

4.1 Introduction 361
4.2 Types 362
4.3 Macros 362
4.4 Structures 362

Index 411

Figures

Figure 1.1 Sample Reference Page 4

Chapter

1

Introduction

1.1 Overview 3

1.2 How to Use This Manual 4

1.2.1 C Format 5

1.2.2 MS OS/2 Include Files 5

1.2.3 MS OS/2 Calling Conventions 6

1.2.4 Bit Masks in Function Parameters 7

1.2.5 Structures 8

1.3 Naming Conventions 8

1.3.1 Parameter and Field Names 8

1.3.1.1 Prefixes 9

1.3.1.2 Base Types 9

1.3.2 Constant Names 11

1.4 Notational Conventions 11

1.1 Overview

This manual describes the system functions of Microsoft® Operating System/2 (MS® OS/2) that are new or modified for version 1.2. These functions let MS OS/2 programs use the operating system to carry out tasks such as reading and writing extended attributes for disk files, creating and using multiple-line entry fields, creating and accessing disk files through installable file systems, and displaying help text in a Presentation Manager application.

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. MS OS/2 programs request operating-system services by calling system functions.

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

Chapter 2, "Overviews," describes the new features and system functions for MS OS/2, version 1.2. This chapter explains the purpose of the functions and gives the operating-system concepts behind them. It also shows how the MS OS/2 system functions work together to carry out specific tasks.

Chapter 3, "Functions and Messages Directory," lists the MS OS/2, version 1.2, system functions and messages. Three categories of functions and messages are included: those that are new for MS OS/2, version 1.2; those that are updated, or changed, from MS OS/2, version 1.1; and those that contain corrections for errors that appeared in the *Microsoft Operating System/2 Programmer's Reference, Volume 2* and *Volume 3*. The category of each item is clearly marked.

This chapter defines the purpose of each function and each message, gives its syntax, describes any parameters, and gives possible return values. Many of the descriptions also show program examples that illustrate how the function or message is used to carry out simple tasks.

Chapter 4, "Types, Macros, Structures," lists and describes the new and updated data types and structures used by MS OS/2, version 1.2, system functions.

This manual is intended to describe the MS OS/2 system functions, messages, types, and structures that are new or that have been modified for MS OS/2, version 1.2. It does not explain how to use these functions to carry out specific tasks. For more information on this topic, see the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

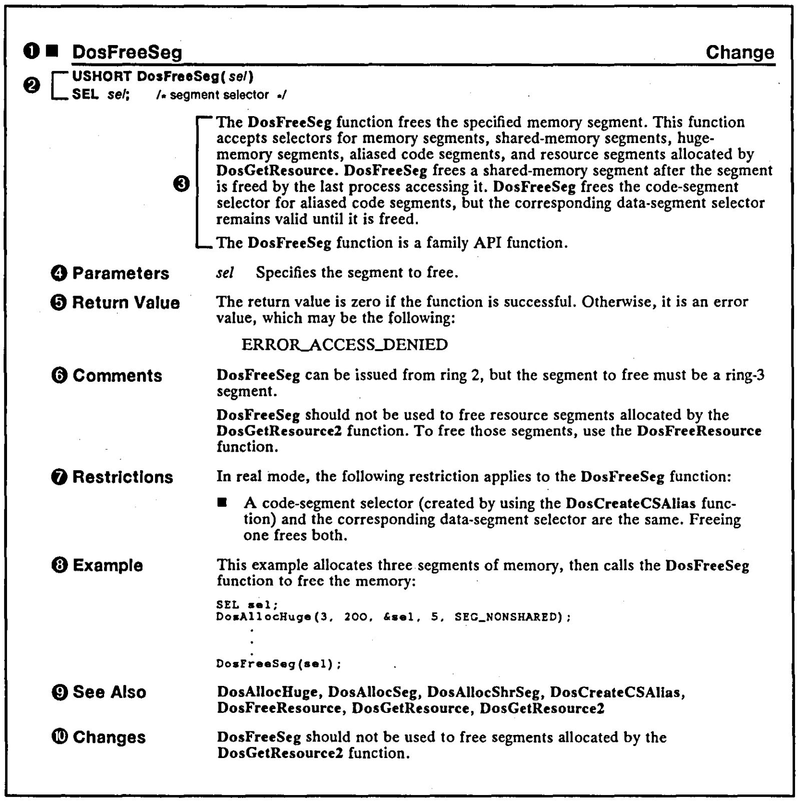
Also, this manual does not fully describe all MS OS/2 base system and Presentation Manager functions. MS OS/2 base system functions enable programs to use the operating system to carry out such tasks as reading from and writing to disk files; allocating memory; starting other programs; and using the keyboard, mouse, and video screen. Presentation Manager functions let programs use the multitasking, window-management, and graphics features of MS OS/2. For more information on MS OS/2 Presentation Manager functions, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2*. For more information on MS OS/2 base system functions, see the *Microsoft Operating System/2 Programmer's Reference, Volume 3.*

In addition, this manual references but does not discuss QuickHelp, the displa program for Microsoft documentation databases. For more information on QuickHelp, see *Microsoft Operating System/2 Getting Started*, available with the Microsoft OS/2 Presentation Manager Toolkit.

1.2 How to Use This Manual

This manual provides detailed information about each MS OS/2, version 1.2, system function, message, and structure. Each item has the format shown in Figure 1.1:

Figure 1.1
Sample Reference Page



❶ ■ **DosFreeSeg** — **Change**

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

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

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

❹ **Parameters** *sel* Specifies the segment to free.

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

ERROR_ACCESS_DENIED

❻ **Comments** **DosFreeSeg** can be issued from ring 2, but the segment to free must be a ring-3 segment.

DosFreeSeg should not be used to free resource segments allocated by the **DosGetResource2** function. To free those segments, use the **DosFreeResource** function.

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

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

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

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

❾ **See Also** **DosAllocHuge, DosAllocSeg, DosAllocShrSeg, DosCreateCSAlias, DosFreeResource, DosGetResource, DosGetResource2**

❿ **Changes** **DosFreeSeg** should not be used to free segments allocated by the **DosGetResource2** function.

These are the elements shown in Figure 1.1:

1 The name of the item and its MS OS/2, version 1.2, status (new, change, or correction). The name of the function, message, or structure appears on the left. Its MS OS/2, version 1.2, status is given on the right.

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

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

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

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

6 General comments about how the function, message, or structure can be used.

7 Restrictions that affect how the function operates in real mode.

8 An example showing how the function or message can be used to accomplish a simple task.

9 A list of related functions and messages.

10 A summary of the item's changes or corrections for MS OS/2, version 1.2.

1.2.1 C Format

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

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

```
/* Play a note for 1 second. */

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

1.2.2 MS OS/2 Include Files

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

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

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

1.2.3 MS OS/2 Calling Conventions

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

- You must push the parameters on the stack. In this manual, each function description lists the parameters in the order they must be pushed. The left parameter must be pushed first, the right parameter last. If a parameter specifies an address, the address must be a far address; that is, it must have the form *selector:offset*. The *selector* must be pushed first, then the *offset*.
- The function automatically removes the parameters from the stack as it returns. This means the function must have a fixed number of parameters.
- You must use an intersegment call instruction to call the function. This is required for all dynamic-link-library functions.
- The function returns a value, possibly an error value, in either the **ax** register or the **dx:ax** register pair. Only the **di** and **si** register values are guaranteed to be preserved by the function. MS OS/2 system functions may preserve other registers as well, but they do not preserve the **flags** register. The contents of the **flags** register are undefined; specifically, the direction flag in the register may be changed. However, if the direction flag was zero before the function was called, it will be zero after the function returns.

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

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

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

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

```
# include <os2.h>

HFILE hfile;
USHORT usAction;

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

1.2.4 Bit Masks in Function Parameters

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

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

1.2.5 Structures

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

1.3 Naming Conventions

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

1.3.1 Parameter and Field Names

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

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

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

1.3.1.1 Prefixes

The following standard prefixes are used in MS OS/2 naming conventions:

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

1.3.1.2 Base Types

The following standard base types are used in MS OS/2 naming conventions:

Base type	Type/Description
f	**BOOL.** A 16-bit flag or Boolean value. The qualifier should describe the condition associated with the flag when it is TRUE. For example, *fSuccess* is TRUE if successful, FALSE if not; *fError* is TRUE if an error occurs and FALSE if no error occurs. For objects of type **BOOL**, a zero value implies FALSE, a nonzero value implies TRUE.
ch	**CHAR.** An 8-bit signed value.

Base type	Type/Description
s	**SHORT**. A 16-bit signed value.
l	**LONG**. A 32-bit signed value.
uch	**UCHAR**. An 8-bit unsigned value.
us	**USHORT**. A 16-bit unsigned value.
ul	**ULONG**. A 32-bit unsigned value.
b	**BYTE**. An 8-bit unsigned value. Same as *uch*.
sz	**CHAR[ ]**. An array of characters, terminated with a null character (the last byte is set to zero).
fb	**UCHAR**. An array of flags in a byte. This base type is used when more than one flag is packed in an 8-bit value. Values for such an array are typically created by using the logical OR operator to combine two or more values.
fs	**USHORT**. An array of flags in a short (16-bit unsigned value). This base type is used when more than one flag is packed in a 16-bit value. Values for such an array are typically created by using the logical OR operator to combine two or more values.
fl	**ULONG**. An array of flags in a long (32-bit unsigned value). This base type is used when more than one flag is packed in a 32-bit value. Values for such an array are typically created by using the logical OR operator to combine two or more values.
sel	**SEL**. A 16-bit value used to hold a segment selector.

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

avldt	*fsinf*	*matlf*	*ptrdd*
cbnd	*fsqbf*	*mlectl*	*sbcd*
dena	*fsts2*	*mlefrd*	*stsdata*
eaop	*fuc*	*mlemrg*	*swblk*
efd	*fur*	*mleovr*	*ti*
fat	*gea*	*mlesrch*	*viocreg*
fea	*geal*	*nmpsmst*	*viofcsz*
feal	*hci*	*param*	*vioin*
findbuf2	*hinit*	*pres*	*viomi*
flc	*ht*	*prfpro*	*viosett*
flr	*kbci*	*progde*	*viosz*
fm	*kbhw*	*progt*	*viouline*
frwc	*ldtaddr*	*progti*	*wprm*
fsc	*lis*	*ptrcbf*	

1.3.2 Constant Names

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

1.4 Notational Conventions

The following notational conventions are used throughout this manual:

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

Chapter

2

Overviews

2.1 Introduction 17

2.2 Installable File Systems 17
2.2.1 About Installable File Systems 17
2.2.1.1 File-System Functions 18
2.2.1.2 File-System Volume 18
2.2.1.3 Local and Remote File Systems 19
2.2.1.4 Pseudo-Character Device 19
2.2.1.5 Filename Conventions 20
2.2.1.6 Filenames in DOS-Compatibility Mode 20
2.2.1.7 Filenames in User Input 20
2.2.1.8 Metacharacters in Filenames 21
2.2.1.9 File-System Errors 22
2.2.2 Summary 22

2.3 Extended Attributes 23
2.3.1 About Extended Attributes 23
2.3.2 Using Extended Attributes 23
2.3.2.1 Naming Conventions 24
2.3.2.2 Data-Type Conventions 24
2.3.3 Standard Extended Attributes 25
2.3.3.1 .TYPE 25
2.3.3.2 .KEYPHRASES 25
2.3.3.3 .SUBJECT 25
2.3.3.4 .COMMENTS 26
2.3.3.5 .HISTORY 26
2.3.3.6 .VERSION 26
2.3.3.7 .ICON 26
2.3.3.8 .ASSOCTABLE 26
2.3.3.9 .HPFSNAME 27
2.3.3.10 Supporting Extended Attributes 27
2.3.3.11 Multivalue Data-Type Fields 27

Chapter
2

2.3.3.12 Multivalue, Multitype Attributes 27
2.3.3.13 Multivalue, Single-Type Attributes 28
2.3.3.14 ASN.1 28
2.3.3.15 Include Extended-Attribute Type 28
2.3.4 Summary 28

2.4 Profile Manager 28
2.4.1 About Profile Manager 28
2.4.2 Using Profile Manager 29
2.4.2.1 Creating or Opening an Initialization File 29
2.4.2.2 Reading and Writing Settings 30
2.4.2.3 Identifying the Initialization Files 30
2.4.2.4 Creating Groups and Program Lists 30
2.4.3 Summary 31

2.5 Help Manager 32
2.5.1 About Help Manager 32
2.5.2 Using Help Manager in Applications 33
2.5.2.1 Creating a Help Instance 33
2.5.2.2 Creating a Help Table 34
2.5.2.3 Creating a Help Library 35
2.5.2.4 Using the F1 Key 35
2.5.2.5 Using the Help Menu 36
2.5.2.6 Using Help Buttons 37
2.5.2.7 Destroying a Help Instance 38
2.5.2.8 Handling Errors 38
2.5.3 Help Hooks and Help Manager 38
2.5.4 Summary 39
2.5.4.1 Functions 39
2.5.4.2 Messages Sent by Help Manager 39
2.5.4.3 Messages Sent to Help Manager 39

2.6 Combination-Box Controls 40
2.6.1 About Combo Boxes 40
2.6.2 Using Combo Boxes 41
2.6.3 Summary 42
2.6.3.1 Combo-Box Styles 42
2.6.3.2 Messages Sent to a Combo Box 42
2.6.3.3 Messages Sent by a Combo Box 42

2.7 Multiple-Line Entry Fields 43
2.7.1 About Multiple-Line Entry Fields 43
2.7.1.1 Editing MLE Text 43

Chapter 2

2.7.1.2 Formatting MLE Text 44
2.7.1.3 Importing and Exporting MLE Text 45
2.7.1.4 Copying and Pasting MLE Text 46
2.7.1.5 Searching and Replacing MLE Text 46
2.7.1.6 MLE Notification Codes 46
2.7.1.7 MLE Styles 47
2.7.2 Using Multiple-Line Entry Fields 47
2.7.3 Summary 48
2.7.3.1 MLE Styles 48
2.7.3.2 Messages Sent to an MLE 48
2.7.3.3 Messages Sent from an MLE 50

2.1 Introduction

This chapter describes the MS OS/2 system functions in individual-topic sections. Each section describes a portion of MS OS/2 that lets an application carry out a specific task or set of related tasks. For example, the section about the multiple-line entry field (MLE) defines basic MLE terms, describes the role of multiple-line entry-field messages, and illustrates how to use those messages.

Each topic section in this chapter gives a general description and programming samples. Each section discusses the purpose and operation of pertinent MS OS/2 functions. The programming samples show how to use those MS OS/2 functions in applications to carry out useful tasks.

In many cases, it is assumed that you have basic knowledge of some other portions of MS OS/2. Each section lists the prerequisites for understanding the concepts and terms described in that section.

2.2 Installable File Systems

This section describes how MS OS/2 enables programs to use installable file systems. A file system is the combination of software and hardware that supports storing information on a storage device. An installable file system is a file system whose software can be installed when the operating system starts. MS OS/2 supports installable file systems and permits users to have multiple file systems active at the same time.

This section also describes some of the MS OS/2 functions that let programs create, read, and write data files in installable file systems. Because installable file systems are not available with releases of MS OS/2 prior to version 1.2 or with MS-DOS®, versions 2.0 through 3.3, programs that use the family application programming interface (family API) cannot use functions that are specific to installable file systems.

2.2.1 About Installable File Systems

In MS OS/2, version 1.2, users install a file system by specifying the file-system components in the *config.sys* file. The file-system software consists of device drivers that access storage devices and dynamic-link libraries that control the format of information on a device and manage the flow of data to and from the device. The user must use the **device** command to specify the device driver and the **ifs** command to specify the dynamic-link library. MS OS/2 loads the device driver and dynamic-link library and initializes a specific device for use as a file system.

MS OS/2, version 1.2, has two file systems: the file allocation table (FAT) file system and the high-performance file system (HPFS). These file systems define how information is organized on the storage devices. Both file systems create data files supported by one or more tables that specify the location and size of the data files on the storage device.

The file allocation table (FAT) file system is the default file system for MS OS/2; it does not need to be installed. The FAT file system, used in previous releases of MS OS/2 and also in MS-DOS, controls storage of data files for fixed and floppy disks. The FAT file system is hierarchical, allowing multiple directories on the disk. Each directory can contain one or more files. The distinguishing feature of the FAT file system is its 8.3 filename convention. Under this convention, the filename consists of a filename (up to eight characters), a separating period (.), and a filename extension (up to three characters).

The high-performance file system (HPFS) is an installable file system for MS OS/2. It is an hierarchical file system and allows for multiple directories. HPFS controls storage of data for fixed disks. Filenames under HPFS can be any practical length and can contain characters that are not valid for the FAT file system, for example, spaces and underscores (_). In many cases, accessing files under HPFS is faster than accessing similar files under the FAT file system.

A user can choose either or both file systems. Programs must be able to work with any file system. Fortunately, MS OS/2 provides a common set of file-system functions that are not dependent upon a particular file system; it also gives guidelines for working with file systems, such as specific filename conventions.

2.2.1.1 File-System Functions

MS OS/2 provides a standard set of file-system functions. This means that programs can create, open, read, write, copy, and delete files and directories by using the same functions regardless of which file system is used. When a program calls a file-system function, MS OS/2 passes the request to the dynamic-link library that supports the file system. Most file-system processing, such as validating filenames, is carried out by the dynamic-link library. If an error occurs, the file system returns the error to MS OS/2, which then passes it back to the calling program.

Occasionally, a file system may extend the standard set of file-system functions by providing file-system control functions. The control functions are specific to the given file system. A program can call a control function by using the **DosFSCtl** function, which directs MS OS/2 to pass the control-function information to the dynamic-link library for the particular file system.

2.2.1.2 File-System Volume

MS OS/2 allows more than one file system on a single storage device. If the device can have more than one logical partition (or volume), each partition can be initialized as an MS OS/2 partition and given a valid MS OS/2 file system. For each volume, MS OS/2 determines the type of file system the first time the volume is accessed by a function or when the media in the drive changes. After that, MS OS/2 manages all input and output to that volume by using the corresponding dynamic-link library for the file system.

MS OS/2 uses the volume label and serial number to ensure that the media in the drive does not change while there are outstanding requests for input and output. Each volume has a volume label and a 32-bit volume serial number, stored in a reserved location in logical sector zero at the time of formatting. If the volume label and serial number do not match, MS OS/2 signals the critical-error handler to prompt the user to insert the volume that has the specified serial number and label. MS OS/2 maintains the connection between the media and the volume label and serial number until all open files on the volume are closed and all search references and cache-buffer references are removed. The system redetermines the type of the file system and the volume label and serial number for the volume only when the media changes.

2.2.1.3 Local and Remote File Systems

Installable file systems work with a variety of storage devices. A file system on a local device such as a disk drive or virtual disk is called a local file system. A file system on a remote device such as a disk drive on another computer is called a remote file system. A program can establish a connection to a local or a remote file system by using the **DosFSAttach** function.

For a local file system, MS OS/2 uses a block device driver to handle input and output to the device. MS OS/2 automatically connects most (if not all) local file systems when it starts. However, a program can connect and disconnect (sometimes called mount and dismount) additional file systems as needed.

For a remote file system, the corresponding device driver typically accesses a communications or network device instead of a block device driver used to access disk hardware. Typically, the actual storage device is located on another computer, and the two computers communicate requests and data through a network connection. A program can connect a remote file system to a drive letter by using the **DosFSAttach** function. Once the connection is made, the program can access directories and files on the remote device simply by using the assigned drive letter, treating the remote device as if it were on the same computer.

2.2.1.4 Pseudo-Character Device

A program can attach a device name to a file system and use the file system as a pseudo-character device (also called a single-file device). Attaching a device name to a file system lets a program open the device associated with the file system as if it were a character device (for example, a serial port) and read from and write to the device by using the **DosRead** and **DosWrite** functions. Unlike with a character device, a program can use the **DosChgFilePtr** and **DosFileLocks** functions for working with a pseudo-character device. An MS OS/2 pseudo-character device name is a null-terminated string in the format of an MS OS/2 filename in a subdirectory called *\dev*.

A file system that can be attached to a pseudo-character device is typically associated with a single disk file or with a special storage device such as a tape drive. The file system establishes a connection with the device and transfers requests and data between MS OS/2 and the device. The following example attaches the device associated with the file system *bcrvmpc1* to the pseudo-character device named *\dev\host*:

```
BYTE bData[];
USHORT cbData;

DosFSAttach("\dev\host", "bcrvmpc1", bData, cbData, 0);
```

If the program successfully attaches the file system, the program can then open the file *\dev\host* by using the **DosOpen** function, read host-created data by using the **DosRead** function, and write data and commands to the host by using the **DosWrite** function. This example assumes that the name *bcrvmpc1* corresponds to an installable file system and that the file system can perform the necessary host communication and translation.

2.2.1.5 Filename Conventions

Filename conventions are the rules used to form names that uniquely identify files in a given file system. Although each installable file system can have specific rules about how individual components in a directory or filename are formed, all file systems follow the same general conventions for combining components. For example, the FAT file system requires that file and directory names have the 8.3 filename format, HPFS allows names to be any length, but both file systems use the backslash (\) character to separate directory names and the filename when forming a path.

When creating names for directories and files or when processing names supplied by the user, programs should follow these general rules:

1 Process a path as a null-terminated string. A program can determine maximum length for a path by using the **DosQSysInfo** function.

2 Use any character in the current code page for a name, but do not use a path separator, a character in the range 0 through 31, or any character explicitly disallowed by the file system. Although a name can contain characters above 127, a program must be able to switch code pages if necessary to access the corresponding file.

3 Compare names using a case-insensitive comparison. Names such as *ABC*, *Abc*, and *abc* are considered to be the same name.

4 Use the backslash (\) and/or the forward slash (/) to separate components in a path. No other character is accepted as a path separator.

5 Use the dot (.) as a directory component in a path to represent the current directory.

6 Use two dots (..) as a directory component in a path to represent the parent of the current directory.

7 Use a period (.) to separate components in a directory name or filename. Unless explicitly defined by a file system, there are no restrictions on the number of components in a name.

2.2.1.6 Filenames in DOS-Compatibility Mode

For compatibility with existing DOS 3.*x* programs, all file systems support the FAT file system's 8.3 filename format. This means that programs running in DOS-compatibility mode can access files on non-FAT file systems if the filenames have the 8.3 format. To guarantee this rule, MS OS/2 automatically applies the 8.3 truncation rules to all filenames given in file-system requests from DOS-compatibility mode.

2.2.1.7 Filenames in User Input

Users often supply filenames as part of a program's command line or in response to a prompt from the program. Traditionally, users have been able to supply more than one filename on the command line or in a prompt by separating the names with certain characters, such as a blank space. In some file systems, however, traditional separators can be used as valid filename characters. This means that some additional conventions are required to ensure that a program processes all characters in a name.

When a program processes arguments (including filenames) from its command line, the program should treat the double quotation mark (") and the caret (^) as quotation characters. All characters between the starting and closing double quotation marks should be processed as a single argument. The character immediately following the caret should be processed as part of the current argument. In both cases, the quotation characters are discarded and not treated as part of the final argument.

When a program processes two or more filenames from a dialog box or other prompt, it expects the user to enter each filename on a new line. For example, a Presentation Manager application should use a multiple-line entry field to prompt for multiple filenames. This makes the use of quotation characters unnecessary.

When a program is started from File Manager, File Manager may construct a command line for the program. If the command line includes filenames, File Manager separates each argument with a space character and marks the end of the argument list with two null characters. Programs that start other programs by using the **DosExecPgm** function also can pass arguments using this convention or by using quotation characters. In practice, most programs receive a command line as a single, null-terminated string. Therefore, programs that use the **DosExecPgm** function should prepare command lines as a single string with any filenames in the string enclosed in quotation marks.

2.2.1.8 Metacharacters in Filenames

To give the user a shortcut to entering long lists of names, programs that accept more than one filename on their command line can allow metacharacters in filenames. The metacharacters, the asterisk (*) and the question mark (?), represent placeholders in a filename. Although a name that contains metacharacters is not a complete filename, a program can use functions, such as **DosFindFirst** and **DosEditName**, to expand the name (replace the metacharacters) to create one or more valid filenames.

A program can expand a name with metacharacters to a list of filenames by using the **DosFindFirst** function. The asterisk (*) matches one or more characters, including blanks. The question mark (?) matches one character, unless that character is a period (.). To match a period, the original name must contain a period.

A program can create a new filename from an existing name by using the **DosEditName** function. This function takes a template (a name with metacharacters) and expands it, using characters from an existing name. An asterisk (*) in the template directs the function to copy all characters in the existing name until it locates a character that matches the character following the asterisk. A question mark (?) directs the function to copy one character unless that character is a period. The period (.) in the template directs the function look for and move to the next period in the existing name, skipping any characters between the current position and the period.

The metacharacters are illegal in all but the last component of a path.

2.2.1.9 File-System Errors

Some MS OS/2 file-system functions return the following errors:

Value	Meaning
ERROR_WRITE_PROTECT	The disk in the drive is write-protected.
ERROR_BAD_UNIT	There is a breakdown of internal consistency in mapping between the logical drive and the device driver. Internal error.
ERROR_NOT_READY	The device is not ready.
ERROR_BAD_COMMAND	There is a breakdown of internal consistency between the expected capability of a device driver and its true capability.
ERROR_CRC	The device driver detected a cyclic redundancy check (CRC) mismatch.
ERROR_BAD_LENGTH	There is a breakdown of internal consistency between the expected length of a request packet and the true length. Internal error.
ERROR_SEEK	The device driver detected an error during a seek operation.
ERROR_NOT_DOS_DISK	The disk is not recognized as being manageable by MS OS/2.
ERROR_SECTOR_NOT_FOUND	The device is unable to find the specific sector.
ERROR_OUT_OF_PAPER	The printer is out of paper.
ERROR_WRITE_FAULT	Other write-specific error.
ERROR_READ_FAULT	Other read-specific error.
ERROR_GEN_FAILURE	Other error.

There are also errors defined by and specific to the specific device driver. These are indicated by either 0xFF or 0xFE in the high byte of the error code.

2.2.2 Summary

The following MS OS/2 file-system functions work with installable file systems:

DosCopy Copies a file or subdirectory.

DosEditName Transforms a source string using an editing string.

DosFileIO Performs file I/O (locking, unlock, seek, read, and write operations).

DosFindFirst2 Finds the first file that matches a specified filename and attributes.

DosFSAttach Attaches or detaches a drive or pseudo-character device from a remote file system.

DosFSCtl Calls file-system functions that are not part of the standard I/O functions.

DosGetResource2 Retrieves a resource for a module.

DosMkDir2 Creates a directory.

DosOpen2 Opens or creates a file with extended attributes.

DosQFSAttach Queries information about an attached file system.

DosSetPathInfo Sets information for a file or directory.

DosShutdown Shuts down the file system.

2.3 Extended Attributes

This section describes how to use extended attributes to store information about your files and directories. Before reading this section, you should be familiar with the MS OS/2 file system.

2.3.1 About Extended Attributes

Extended attributes can be thought of as a list of facts attached to a file or directory. MS OS/2 stores extended attributes separate from the file or directory so that the attributes do not affect the contents of the file or directory. An application uses extended attributes to provide a description of the file or directory, but does not place the description in the file or directory itself.

Each extended attribute has two parts: a name and a value. The name is a null-terminated string; applications can choose any convenient name. The value is corresponding data; it can be text, a bitmap, or any binary data. The application that creates the extended attributes and the applications that read the extended attributes must recognize the format and meaning of the data associated with a given name.

2.3.2 Using Extended Attributes

Applications can examine, add, and replace extended attributes at any time. The **DosOpen2** function adds extended attributes to new or existing files; the **DosMkDir2** function adds extended attributes to new directories. Any application can read the extended attributes by using the **DosQFileInfo** or **DosQPathInfo** function. Applications can also search for files that have specific extended attributes by using the **DosFindFirst** and **DosFindNext** functions.

A file can have any number of extended attributes. Each extended attribute can be up to 64K long. For MS OS/2, version 1.2, the sum of all extended attributes for a file must not exceed 64K.

2.3.2.1 Naming Conventions

Although an application can choose any name for the extended attributes it creates, other applications cannot read the extended attributes unless they also recognize the corresponding format. Because many applications use extended attributes consisting of text, bitmaps, and other similar data, a set of names has been adopted to help identify these formats when used in extended attributes. An application need not be limited to this set of standard extended attributes, but should use it as a way for many applications to access a common set of information.

The names for all standard extended attributes use a dot (.) as a prefix. The leading dot is considered reserved, so no application should define extended attributes that start with a dot. Also, extended attributes that start with the characters $, @, &, and + are reserved for system use. To ensure that its extended attributes are unique, an application should use the vendor and application name as a prefix for application-specific extended attributes. For example, Microsoft Excel would use MS EXCEL.MYSTUFF, MS EXCEL.MORESTUFF, and so forth.

2.3.2.2 Data-Type Conventions

Extended attributes can contain any type of data. To identify the type of information, the first word of extended-attribute data should specify one of the following data types:

Value	Meaning
EAT_BINARY	Binary data; the first word specifies length.
EAT_ASCII	ASCII text; the first word specifies length.
EAT_BITMAP	Bitmap data; the first word specifies length.
EAT_METAFILE	Metafile data; the first word specifies length.
EAT_ICON	Icon data; the first word specifies length.
EAT_EA	ASCII name of associated data; the first word specifies length.
EAT_MVMT	Two or more consecutive extended-attribute values; each value has a explicitly specified type.
EAT_MVST	Two or more consecutive extended-attribute values; all values have the same type.
EAT_ASN1	ASN.1 field data.

In all cases, the length specifies the number of bytes of data. Other values for data types, in the range 0x0000 through 0x7FFF, can be used for user-defined extended attributes. User-defined data should also specify the length.

For example, here is how to represent the string "Hello":

```
EAT_ASCII                 0005                Hello
```

2.3.3 Standard Extended Attributes

The standard extended attributes are listed in the following sections. The field format follows the data-type conventions given previously. A field can be a multivalue or single-value field.

2.3.3.1 .TYPE

The .TYPE extended attribute indicates the type of file. It is similar to the earlier use of filename extensions. The following file types are predefined:

Plain Text
OS/2 Command File
DOS Command File
Executable
Metafile
Bitmap
Icon
Binary Data
Dynamic Link Library
C Code
Pascal Code
BASIC Code
COBOL Code
FORTRAN Code
Assembler Code
Library
Resource File

Applications can use their own type names, such as Microsoft Excel Chart. The first words in the type name should be the name of the vendor and the application. For example, Microsoft Excel Chart, Microsoft Excel Worksheet, Lotus 1–2–3 Spreadsheet.

Entries should be ASCII. Case is important.

The performance of extended attributes is dependent on the file system. Because some file systems store extended attributes in first-in/first out (FIFO) order, it is important to write the .TYPE entry first so that File Manager can access that information quickly.

2.3.3.2 .KEYPHRASES

The .KEYPHRASES extended attribute specifies text key phrases for the file. Such phrases can be used for a database-style search or to help the user understand the nature of the file.

If there is more than one key phrase, each should be stored in a separate entry in a multivalue field. Each entry should be ASCII.

2.3.3.3 .SUBJECT

The .SUBJECT extended attribute contains a brief summary of the file's content and/or purpose. This attribute should be less than 40 characters long.

This field should be a single-value ASCII entry.

2.3.3.4 .COMMENTS

The .COMMENTS extended attribute contains miscellaneous notes about the file. It can be a multivalue field and be of any type. This field is intended as a reminder note. For example, it could contain some notes about the intent of a file or a picture.

2.3.3.5 .HISTORY

The .HISTORY extended attribute lists the history of a file's modification. It lists the author of the file and all subsequent changes. Each action entry should be a separate field in a multivalue field. Each entry should be ASCII.

The application can let the user decide when an entry is placed into the history field, to avoid unnecessary file growth. For example, there are some cases when it is important to note when a document is printed; however, it is probably unnecessary to note every time the file was printed.

2.3.3.6 .VERSION

The .VERSION extended attribute is a version number of the file format (for example, Excel Worksheet 1).

This attribute should be ASCII or binary. It should be modified only by the application. This attribute can also be used to indicate an application or dynamic-link library version.

2.3.3.7 .ICON

The .ICON extended attribute specifies the icon to be used for the file representation, whether in File Manager or when minimized. File Manager can use the .TYPE entry to determine the default application to run and to determine the default icon for that type of file. If there is a .ICON entry, however, it is used instead of the icon associated with a particular type.

If the data type is for an icon, the icon data follows. It is best to provide as much icon information as possible. Ideally, an icon should be 64-by-64 bits in 8-color, device-independent format.

Executable files should simply store the binary icon data in this extended attribute. They should use the .ASSOCTABLE extended attribute to install icons for data files.

2.3.3.8 .ASSOCTABLE

The .ASSOCTABLE extended attribute contains association data for a file. It is created by the Microsoft Operating System/2 Resource Compiler (**rc**), from a table with the following form:

```
ASSOCTABLE assoctable -id
BEGIN
        "type name", "extension", [flags], [icon filename]
        .
        .
END
```

The .ASSOCTABLE extended attribute contains information that associates icons with the data files an application creates. The file-association table associates icons by data type.

The .ASSOCTABLE extended attribute allows an application to indicate the type, extension, and icon for the data files it recognizes. It also contains an ownership flag. This data can be installed automatically by File Manager.

For example, the table for Microsoft Excel could be:

```
"MS Excel Worksheet", "XLS", AF_DEFAULTOWNER, excel.sheet.icon
"MS Excel Chart", "XLC", AF_DEFAULTOWNER, excel.chart.icon
```

The flag entry indicates if the application owns or merely recognizes the type. The icon file contains an icon for that data type.

2.3.3.9 .HPFSNAME

The .HPFSNAME attribute is used when an application attempts to write a file with a long name to a file system that does not support long names. The application should generate a unique short name for the file and notify the user of the new short name. It should then save the original (long) name in the .HPFSNAME extended attribute.

When a file is copied from a system that uses short names to a system that uses long names, the application should check the .HPFSNAME extended attribute. If a value is present, the application should allow the file to be renamed to a long name. The .HPFSNAME extended attribute should then be removed.

2.3.3.10 Supporting Extended Attributes

To support extended attributes, applications should do the following:

1. Fill in the .ASSOCTABLE extended attribute for all major file types that the application recognizes or uses.
2. Fill in the .ICON extended attribute for executable files.
3. Set the .TYPE field for data files it creates.
4. Fill in and use the .HPFSNAME extended attribute as appropriate.
5. Support .HISTORY and .VERSION.
6. Support the other standard extended attributes as appropriate.

2.3.3.11 Multivalue Data-Type Fields

In many cases, extended attributes need to store more than a single piece of information. For example, an extended attribute can store a list of names of people to whom a mail document was sent. The multivalue formats specify how individual pieces of data are stored.

In a multivalue field, the first entry in the list is assumed to be the default. For example, suppose the .TYPE entry contains Text and C Code. Text is the default type. If C Code is the first entry in the list (C Code and Text), then C Code is the default type.

2.3.3.12 Multivalue, Multitype Attributes

The EAT_MVMT type allows a single extended attribute to contain several pieces of information; each piece of information can be a different type.

2.3.3.13 Multivalue, Single-Type Attributes

The EAT_MVST type sets up a multivalue field in which each piece of information is of the same type.

2.3.3.14 ASN.1

The EAT_ASN1 type is an ISO standard for describing multivalue data streams.

2.3.3.15 Include Extended-Attribute Type

The EAT_EA type indicates that the data is continued in another extended-attribute entry associated with the file. Among other things, this allows for extended attributes greater than 64K (but not exceeding the limit per file).

2.3.4 Summary

The following MS OS/2 functions create and manage extended attributes:

DosFindFirst2 Finds the first file that matches the specified filename and attributes.

DosMkDir2 Creates a directory.

DosOpen2 Opens or creates a file with extended attributes.

DosQFileInfo Retrieves file information, including 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, its size, and its attributes. It also returns information about a file's extended attributes.

DosQPathInfo Retrieves information about a file or directory.

DosSetFileInfo Sets information about a file, including 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. It can also set extended attributes for a file.

DosSetPathInfo Sets information for a file or directory.

2.4 Profile Manager

This section describes how to use the MS OS/2 Profile Manager to store and retrieve information about your application and the system from the MS OS/2 initialization files. Before reading this section, you should be familiar with the MS OS/2 initialization files.

Profile Manager functions replace the MS OS/2 initialization-file functions described in the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

2.4.1 About Profile Manager

Profile Manager enables applications to create their own initialization files and to access the MS OS/2 initialization files, *os2.ini* and *os2sys.ini*. An initialization file is a convenient place to store information between sessions. Just as MS OS/2 uses the *os2.ini* and *os2sys.ini* files to store configuration information for when it starts, an application can create initialization files that store information it uses to initialize windows and data when it starts.

Because all initialization files are binary files, the user cannot view or edit them directly. A file consists of one or more sections; each section contains one or more settings, or keys. Each key consists of two parts: a name and a value. Both section names and key names are null-terminated strings. A key value can be a null-terminated string, a null-terminated string representing a signed integer, or individual bytes of data.

The MS OS/2 initialization files, *os2.ini* and *os2sys.ini*, contain sections and settings used by the MS OS/2-system applications (such as Desktop Manager, Control Panel, and Print Manager). Although applications can read settings from the MS OS/2 initialization files, only rarely will an application need to change a setting. One common task that does change the settings in the MS OS/2 initialization files is adding a group and program list to Desktop Manager. For example, the installation program for an application can create a new group for the application and its related utilities by using Profile Manager functions.

Once an initialization file is created, an application can rename, copy, move, and delete the file just like any other file. Although an application can also read and write to the file as if it were a binary file, the application should always use Profile Manager functions to access the contents of the file.

2.4.2 Using Profile Manager

You can use Profile Manager functions in character-based MS OS/2 programs as well as in Presentation Manager applications. A thread that calls Profile Manager functions must have initialized an anchor block by using the **WinInitialize** function. You create an initialization file or open an existing one by using the **PrfOpenProfile** function. You then store and retrieve information from the file by using functions such as **PrfQueryProfileString** and **PrfWriteProfileString**. You can also create and manage groups and program lists by using functions such as **PrfAddProgram** and **PrfCreateGroup**.

2.4.2.1 Creating or Opening an Initialization File

You can create an initialization file or open an existing initialization file by using the **PrfOpenProfile** function. The function takes a handle to an anchor block and a pointer to the name of an initialization file. If the file doesn't exist in the given path, the function automatically creates an initialization file.

The following example creates an initialization file named *pmtools.ini* in the current directory:

```
HAB hab;
HINI hini;

hab = WinInitialize(0);
if ((hini = PrfOpenProfile(hab, "pmtools.ini")) == NULL)
    /* initialization file not opened or created */
```

If it is successful, the **PrfOpenProfile** function returns a handle to the initialization file. Otherwise, it returns NULL. Once you have an initialization-file handle, you can create new sections in the file and make new settings.

To close an initialization file, you use the **PrfCloseProfile** function.

2.4.2.2 Reading and Writing Settings

You can read and write strings, integers, and binary data to and from an initialization file. To read from or write to an initialization file, you must provide a section and a key name that specifies which setting to read or to change. When writing, if there is no corresponding section and/or key name, the section and/or key name is added to the file and assigned the given value.

The following example creates the section "MyApp" and the key name "MainWindowColor" in a previously opened initialization file and assigns the value of the RGB structure to the new setting:

```
HINI hini;
RGB rgb = { 0xff, 0x00, 0x00 };

PrfWriteProfileData(hini, "MyApp", "MainWindowColor", &rgb, sizeof(RGB));
```

To read a setting, you can retrieve the size of the setting and then read the setting into an appropriate buffer by using the **PrfQueryProfileSize** and **PrfQueryProfileData** functions, as shown in the following example. This example reads the setting "MainWindowColor" from the "MyApp" section only if the size of the data is equal to the size of the RGB structure.

```
HINI hini;
ULONG cb;
RGB rgb;

PrfQueryProfileSize(hini, "MyApp", "MainWindowColor", &cb);
if (cb==sizeof(RGB))
    PrfQueryProfileData(hini, "MyApp", "MainWindowColor", &rgb, &cb);
```

You can also read strings by using the **PrfQueryProfileString** function and write strings by using the **PrfWriteProfileString** function. You can read integers (stored as strings) by using the **PrfQueryProfileInt** function.

2.4.2.3 Identifying the Initialization Files

You can retrieve the names of the MS OS/2 initialization files by using the **PrfQueryProfile** function. Although the MS OS/2 initialization files are usually named *os2.ini* and *os2sys.ini*, a user can use other files when starting the system.

The following example retrieves the names of the MS OS/2 initialization files and copies the names of the initialization files to the arrays szUserName and szSysName. Once you know the names of the MS OS/2 initialization files, you can use that name to open the files and read settings.

```
char szUserName[80];
char szSysName[80];
PRFPROFILE prfpro = { 80, (PSZ) szUserName, 80, (PSZ) szSysName };

PrfQueryProfile(hini, &prfpro);
```

You can change the MS OS/2 initialization files to files of your choice by using the **PrfReset** function. This function takes the names of two initialization files and uses them as replacements for the *os2.ini* and *os2sys.ini* files. The system is then reset using the settings in the new files.

2.4.2.4 Creating Groups and Program Lists

You can create a group and a list of programs by using the **PrfCreateGroup** and **PrfAddProgram** functions. A group is a window, managed by Desktop Manager, that contains a list of programs. The user can start a program in the list by selecting the program title or double-clicking the title using the mouse.

The following example creates a new group, named "My Application," and adds one program to it:

```
HPROGRAM hGroup;
HPROGRAM hProg;
PROGDETAILS pprogde;

progde.Length =          sizeof(PROGDETAILS);
progde.progt.progc =     PROG_PM;               /* Prof. Mngr. prog. */
progde.progt.fbVisible = SHE_VISIBLE;           /* visible           */
progde.pszTitle =        "My Application";      /* program title     */
progde.pszExecutable =   "c:\os2\myapp.exe"     /* path to exe file  */
progde.pszStartupDir =   "c:\os2";              /* work directory    */
progde.pszIcon =          "";                   /* empty if not used */
progde.pszEnvironment =   "";
progde.pszParameters =    "";
progde.swpInitial.fs =    0;
progde.swpInitial.cx =    0;
progde.swpInitial.cy =    0;
progde.swpInitial.x =     0;
progde.swpInitial.y =     0;
progde.swpInitial.hwndInsertBehind = NULL;
progde.swpInitial.hwnd =  NULL;

hGroup = PrfCreateGroup(HINI_USER, "My Application", SHE_VISIBLE);
hProg = PrfAddProgram(HINI_USER, &progde, hGroup);
```

2.4.3 Summary

Profile Manager functions open and modify the MS OS/2 initialization files. Note that these functions are new with MS OS/2, version 1.2, and replace the **Win** initialization-file functions in previous versions of MS OS/2.

PrfAddProgram Adds a program title to Desktop Manager.

PrfChangeProgram Replaces information in the program list.

PrfCloseProfile Closes a profile file.

PrfCreateGroup Creates a new program group in a program list.

PrfDestroyGroup Removes a group from Desktop Manager.

PrfOpenProfile Opens a profile file.

PrfQueryDefinition Retrieves program information.

PrfQueryProfile Retrieves profile filenames.

PrfQueryProfileData Retrieves information from the profile file.

PrfQueryProfileInt Retrieves an integer from the profile file.

PrfQueryProfileSize Retrieves the size of data stored at a specified location in the profile file.

PrfQueryProfileString Retrieves a string from the profile file.

PrfQueryProgramCategory Retrieves the program type.

PrfQueryProgramHandle Retrieves program handles that match the name of a specified executable file.

PrfQueryProgramTitles Retrieves information about programs in a group.

PrfRemoveProgram Removes a program from Desktop Manager.

PrfReset Resets Presentation Manager.

PrfWriteProfileData Places binary data in the profile file.

PrfWriteProfileString Places a string in the profile file.

2.5 Help Manager

This section describes how to use Help Manager in MS OS/2 to display help information about your application to the user. Before reading this section, you should be familiar with the Help Manager user interface, messages and message queues, and menus.

Help Manager functions and messages replace the help messages and help hook described in the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

2.5.1 About Help Manager

You use Help Manager to create help panels and to manage user requests for help. A help panel is one or more lines of text that describe some feature of the application. The help panels for an application are stored in compressed format in a help library. The help library is a separate disk file rather than a resource within in the application's executable file. This makes it easy to update a help library or to replace it with international versions of help.

The user requests help in one of three ways: by pressing the F1 key, by using the Help menu, or by clicking the Help button in a dialog box or message box. The application must provide the Help menu and Help buttons in the application, and it must identify a specific help panel for each command or button. When the user requests help, Help Manager displays a help window alongside the application window and fills the help window with the text of the corresponding help panel. The user can view additional help panels in the help window by using the commands in this help window, or dismiss the help window and return to the application.

While the user views help panels, Help Manager processes all user input, notifying the application of actions carried out for or requested by the user. For example, the user can search for, print, or copy help panels using commands from menus in the help window. Help Manager carries out these actions without assistance from the application. In some cases, Help Manager sends a message to the application window so that the application can determine what additional action to take. For example, if the user input results in an error, Help Manager sends an HM_ERROR message to the application.

Help Manager supports hypertext fields—words or phrases in one help panel that refer to other help panels. The user directs Help Manager to display the other help panels by choosing the hypertext field (using either the mouse or keyboard). Hypertext fields can also direct Help Manager to display help panels from other help libraries and even to start other programs. For example, a hypertext field can direct Help Manager to send a message to the application window to start the application tutorial.

You create help libraries by using the Information Presentation Facility Compiler (IPFC). This compiler produces the compressed help library from the text files that contain your help text. The help text consists of actual text and embedded information tags. The information tags direct the compiler to carry out specific actions, such as setting the help-panel name and ID, setting the font and/or color of the text, displaying text in special formats such as lists or tables, adding a bitmap to the panel, and including help text from another file. For more information about the Information Presentation Facility Compiler, you must use QuickHelp, the display program for Microsoft documentation databases, described in *Microsoft Operating System/2 Getting Started*. The Information Presentation Facility Compiler is available only in the Microsoft OS/2 Presentation Manager Toolkit, version 1.2.

2.5.2 Using Help Manager in Applications

In an application, a user should have three ways to access help: by pressing the F1 key, by choosing commands from the Help menu, and by clicking a Help button. Help Manager provides support for all three methods. The following sections explain how to enable this support for your application.

2.5.2.1 Creating a Help Instance

An application can create an instance of Help Manager by using the **WinCreateHelpInstance** function. This function installs a help hook, initializes Help Manager for help processing, and returns a help-instance window handle. The application uses the help-instance window handle to direct Help Manager to carry out requests for help.

To create a help instance, the application first fills a **HELPINIT** structure with information about the help table, the title of the help window, and the help library for the help instance. In the following example, the *helpinit* parameter is the **HELPINIT** structure. The *hab* parameter is the anchor-block handle of the application, returned by the **WinInitialize** function.

```
HAB hab;
HWND hwndHelp;
HELPINIT helpinit = {
    sizeof(HELPINIT),                /* count of bytes in structure   */
    OL,                              /* return value from Help Mngr.  */
    NULL,                            /* pointer to tutorial name      */
    MAKELONG(MY_RESOURCES, OxFFFF)   /* resource ID for help table    */
    NULL,                            /* handle to help table          */
    NULL,                            /* handle to replacement menu    */
    O,                               /* replacement accelerator ID    */
    O,                               /* replacement menu ID           */
    "My Help!",                      /* help-window title             */
    CMIC_HIDE_PANEL_ID,              /* display help title only       */
    "c:\os2\help\myhelp.hlp"         /* path to help library          */
    };

hwndHelp = WinCreateHelpInstance(hab, &helpinit);
```

The application must associate the help instance with a window by using the **WinAssociateHelpInstance** function. This association tells Help Manager which help instance to use when the user requests help in the window or in any of that window's child or owned windows. A help instance can be associated with any frame window (that is, any window created with the WC_FRAME class). The application always can retrieve the handle of the associated window for a help instance by using the **WinQueryHelpInstance** function.

The user requests help by pressing the F1 key, by choosing a command from the Help menu, or by clicking a Help button. These actions cause MS OS/2 to send a WM_HELP message to an application window procedure. To enable Help Manager to process the message and display help, the window procedure should pass the WM_HELP message to the **WinDefWindowProc** or **WinDefDlgProc** function. Although most window procedures immediately pass the WM_HELP message to the **WinDefWindowProc** or **WinDefDlgProc** function, a window procedure can carry out some processing of the WM_HELP message before it passes the message, as shown in the following example. In all cases, however, the window procedure must return the value returned by **WinDefWindowProc** or **WinDefDlgProc**.

```
case WM_HELP:
    /* Preprocess the message here. */
    return (WinDefWindowProc(hwnd, msg, mp1, mp2));
```

2.5.2.2 Creating a Help Table

A help table is a list of window IDs and corresponding help-panel IDs. For each help request, Help Manager uses a help table to translate into a panel ID the window ID given with the request for help. Every help instance must have a help table.

The application must create the help table and associate the help table with the help instance. An application creates a help table by defining it in a resource script file or by initializing a **HELPTABLE** structure. Most applications define the help table in the resource script file, using the **HELPTABLE** and **HELPSUBTABLE** statements as follows:

```
HELPSUBTABLE MY_MAIN_WINDOW_HELP
BEGIN
    HELPSUBITEM  IDM_HELPFORHELP,   IDH_FORHELP
    HELPSUBITEM  IDM_EXTENDEDHELP,  IDH_FOREXTENDED
    HELPSUBITEM  IDM_KEYSHELP,      IDH_KEYS
    HELPSUBITEM  IDM_HELPINDEX,     IDH_INDEX
    HELPSUBITEM  IDM_ABOUT,         IDH_ABOUT
END

HELPSUBTABLE MY_DIALOG_HELP
BEGIN
    HELPSUBITEM  MY_DIALOG,       IDH_DLG_EXTENDED
    HELPSUBITEM  MY_DIALOG_EDIT,  IDH_DLG_EDIT
END

HELPTABLE MY_MAIN_WINDOW
BEGIN
    HELPITEM MY_MAIN_WINDOW, MY_MAIN_WINDOW_HELP, IDH_EXTENDED
    HELPITEM MY_DIALOG,      MY_DIALOG_HELP,      IDH_DLG_EXTENDED
END
```

In the preceding example, the **HELPTABLE** statement defines the help table. It specifies help for two windows: the main window and a dialog window. (The MY_MAIN_WINDOW and MY_DIALOG constants, defined elsewhere, must be unique and must be equal to the window IDs for these given windows.)

The **HELPITEM** statements within the **HELPTABLE** statement identify the main and dialog windows and the help subtables that apply to them. A help subtable specifies the help-panel ID that corresponds to a window ID. The **HELPITEM** statements also specify the help-panel ID for the extended help associated with each window. For example, the dialog window has the help subtable MY_DIALOG_HELP and the extended help panel IDH_DLG_EXTENDED (the MY_DIALOG_HELP and IDH_DLG_EXTENDED constants must be defined elsewhere).

The **HELPSUBTABLE** statements define the window IDs and corresponding help-panel IDs for each child window of the specified main or dialog window.

After receiving a help request, Help Manager determines which window is active and uses the ID of the active window to select a help subtable. Help Manager then determines the ID of the window that has the input focus (if any) and uses the ID of the focus window with the selected help subtable to identify the help panel. After Help Manager identifies the help panel, it displays the help panel in the help window. Help Manager positions the help window next to the "relative" window (the relative window is the window next to which the system displays the help window). The relative window is usually the active window, but it can be set to another window by using the HM_SET_ACTIVE_WINDOW message.

2.5.2.3 Creating a Help Library

You create a help library by using a text editor to create a help text file and then compiling the help text file with the Information Presentation Facility Compiler (IPFC). The help library must contain one or more help panels, each with a unique panel ID or name. In the help text file, each help panel must start with the **:h1** tag. The help text file itself must start with the **:userdoc.** tag and end with the **:euserdoc.** tag. The following help text file contains two help panels:

```
:userdoc.
:h1 res=1.Extended Help
Display this help when the user requests extended help.
:h1 res=2.Other Help.
Display this help when the user requests any other help.
:euserdoc.
```

The **res=** option with the **:h1** tag identifies the panel ID for the help panel. The text immediately following the **:h1** tag specifies the title of the panel. For example, "Extended Help" is the title of the first panel and "Other Help" is the title of the second. All subsequent text, up to the next **:h1** tag, belongs to that help panel.

2.5.2.4 Using the F1 Key

The F1 key is the system Help key. Help Manager automatically enables this key for a window whenever an application creates a help instance and associates it with the frame window. The user can display help for specific items in the window, such as menu commands, by selecting the item and pressing the F1 key. Whenever the user presses the F1 key, Help Manager retrieves the ID of the selected item and uses the ID to locate the corresponding help-panel ID. If Help Manager finds a help-panel ID, it displays that help panel. Otherwise, it displays the extended help panel.

Although Help Manager carries out all processing for the F1 key, the application must provide appropriate help-table entries for each item that can be selected. If the active window is not directly associated with a help instance, Help Manager checks the window's parent and owner windows until it finds an associated help instance. It first checks the parent window, the parent window of the parent window, and so on, until it finds a window that has an associated help instance. Help Manager checks the owner window only if the parent-window check ended at the desktop and no help instance was found.

2.5.2.5 Using the Help Menu

The Help menu lets the user view general help for an application. The menu appears as the last (rightmost) menu in the menu bar and contains the following commands:

Command	Description
Help for Help	Displays general information about help and how to access help.
Extended Help	Displays information about the application window. This help information can explain the fields in the window, the window's purpose, and how the user should interact with the window.
Keys Help	Displays a list of the function keys used by the application.
Help index	Displays an alphabetical list of all the help-index entries for the application. The author of the help text source file creates the help index by including index tags within the help file.
About	Displays copyright information for the application. The About command is used only in the Help menu for the application window.

The application must create the Help menu, add it to the menu bar, and process the menu commands. The most convenient way to create the Help menu and add it to the menu bar is to place the following statements in the application's **MENU** statement in the resource script file:

```
SUBMENU "~Help", 1
BEGIN
    MENUITEM "~Help for Help...",     IDM_HELPFORHELP
    MENUITEM "~Extended Help...",     IDM_EXTENDEDHELP
    MENUITEM "~Keys Help...",         IDM_KEYSHELP
    MENUITEM "Help ~index...",        IDM_HELPINDEX
    MENUITEM SEPARATOR
    MENUITEM "A~bout...",             IDM_ABOUT
END
```

You can assign any values for the IDM_ constants (IDM_HELPFORHELP and IDM_EXTENDEDHELP, for example) as long as the values are unique within the menu.

To process the menu commands, the window procedure for the application must process the WM_COMMAND message. The application receives a WM_COMMAND message whenever the user chooses one of the menu commands. For each Help-menu command, the application must send an appropriate help message to the help instance for the application, as shown in the following statements.

```
case WM_COMMAND:
    switch (SHORT1FROMMP(mp1)) {
    case IDM_HELPFORHELP:              /* display help for help panel */
        WinSendMsg(hwndHelp, HM_DISPLAY_HELP,
                             MPFROMSHORT(IDH_HELPFORHELP),
                             MPFROMSHORT(HM_RESOURCEID));
        break;
    case IDM_EXTENDEDHELP:             /* display extended help       */
        WinSendMsg(hwndHelp, HM_EXT_HELP, OL, OL);
        break;
    case IDM_KEYSHELP:                 /* display keys help panel     */
        WinSendMsg(hwndHelp, HM_KEYS_HELP, OL, OL);
        break;
    case IDM_HELPINDEX:                /* display help index          */
        WinSendMsg(hwndHelp, HM_HELP_INDEX, OL, OL);
        break;
    case IDM_ABOUT:                    /* create about dialog box     */
        WinDlgBox(HWND_DESKTOP, hwnd, MyAboutProc,
            NULL, MY_ABOUTBOX, NULL);
        break;
    }
    return (OL);
```

In the preceding statements, the HM_DISPLAY_HELP message directs Help Manager to display the specific help panel. You can identify the panel by using a panel ID or by using a panel name. In this example, the constant HM_RESOURCEID directs Help Manager to locate the panel using the panel ID, IDH_HELPFORHELP.

The HM_EXT_HELP message directs Help Manager to display extended help for the help instance. The panel ID for extended help is specified in the help table of the help instance. When Help Manager receives HM_EXT_HELP, it uses the extended help-panel ID to locate and display extended help.

The HM_KEYS_HELP message directs Help Manager to display the help panel that contains a description of the application keys. Although the application must supply the panel ID for keys help, the HM_KEYS_HELP message does not take parameters. Instead, whenever Help Manager receives this message, it sends the HM_QUERY_KEYS_HELP message back to the application. The application must return the keys-help panel ID as shown in the following statements:

```
case HM_QUERY_KEYS_HELP:
    return (IDH_KEYSHELP);
```

The HM_HELP_INDEX message directs Help Manager to display the help index for the help instance. Because the help index has no explicit panel ID, this is the only way to display the help index from the application.

Although the About command is usually placed in the Help menu, Help Manager does not support the About command. The application can use the **WinDlgBox** function to display a dialog box that contains copyright information in response to the user choosing the About command. A corresponding dialog template must be defined in the resource script file.

2.5.2.6 Using Help Buttons

Help buttons provide an alternative way to display contextual help for fields in dialog boxes. A Help button is a push button that displays help information when the user clicks it using the mouse. It usually appears in the lower-right part of a dialog box. Clicking a Help button has the same effect as pressing the F1 key (that is, it displays information about the selected field).

The application must add Help buttons to dialog boxes, but Help Manager carries out the processing. The most convenient way to add a Help button to a dialog box is to use a **PUSHBUTTON** statement in the dialog template in the resource script file. The following statements define a very simple dialog box with a Help button:

```
DLGTEMPLATE MY_DIALOG
BEGIN
    DIALOG "My Dialog!", MY_DIALOG, 0,0, 200,85,,FCF_TITLEBAR
    BEGIN
    LTEXT "Enter name:", MY_LABEL, 10,40, 60,15
    ENTRYFIELD "", MY_DIALOG_EDIT, 70,40, 120,15, ES_MARGIN
    DEFPUSHBUTTON "OK", MY_OK, 10,10, 60,15
    PUSHBUTTON "~Help", MY_HELP, 110,10, 60,15,
        BS_NOPOINTERFOCUS|BS_HELP
    END
END
```

The Help button must have the BS_HELP and BS_NOPOINTERFOCUS styles. When the button has the BS_HELP style, the system interprets a button click as a request for help. When the button has the BS_NOPOINTERFOCUS style, the input focus does not move from the Help button when it is clicked; this allows Help Manager to determine which field in the dialog box is selected.

2.5.2.7 Destroying a Help Instance

When a help instance is no longer needed, you can destroy it by using the **WinDestroyHelpInstance** function. This function closes the help-instance window and removes the corresponding help hook. Before destroying the help instance, you should disassociate the help instance from the window by using the **WinAssociateHelpInstance** function and specifying a NULL window handle. After a help instance is disassociated, it can be destroyed.

2.5.2.8 Handling Errors

Help Manager functions typically indicate errors by returning FALSE. If a function is unsuccessful, the application can use the **WinGetLastError** function to retrieve the value of the error.

If the user is viewing a help panel when an error occurs, Help Manager sends the HM_ERROR message to the active application window to notify the application of the error. Help Manager does not display error messages to the user; the application must display its own messages.

2.5.3 Help Hooks and Help Manager

Help Manager installs a help hook when the application creates the Help Manager instance. This hook enables Help Manager to trap user requests for help. When using Help Manager for your application, it is recommended that you do not install your own help hooks. If you choose to do so, however, you must install the help hook prior to creating the help instance because the Help Manager help-hook procedure always returns TRUE, preventing all subsequent hook procedures from being called. If you do install a help hook, it must return FALSE so that Help Manager can process requests for help.

2.5.4 Summary

The following MS OS/2 functions and messages work with Help Manager.

2.5.4.1 Functions

MS OS/2 provides the following help functions:

WinAssociateHelpInstance Associates a help instance with a given window.

WinCreateHelpInstance Creates a help instance.

WinCreateHelpTable Identifies or changes the pointer to the help table.

WinDestroyHelpInstance Destroys an instance of Help Manager.

WinLoadHelpTable Identifies or changes the handle of the module that contains the help-table resource and the ID of that resource.

WinQueryHelpInstance Retrieves the handle of the help instance associated with the specified window.

2.5.4.2 Messages Sent by Help Manager

Help Manager sends the following messages to the application:

HM_ACTIONBAR_COMMAND Sent to the application when the user chooses a command from an application-supplied menu.

HM_ERROR Notifies the application of an error caused by a user action.

HM_EXT_HELP_UNDEFINED Notifies the application that an extended help panel is not defined for the active window.

HM_HELPSUBITEM_NOT_FOUND Sent to the application when the user requests help about a field and the system cannot find a related entry in the help subtable.

HM_INFORM Notifies the application that the user has selected a hypertext field in the help window. The hypertext field must have been created using the **:inform** tag.

HM_QUERY_KEYS_HELP Sent to the application when the user requests keys help. The application responds by returning the ID of the requested keys-help panel.

HM_TUTORIAL Sent to the application when the user chooses the Tutorial command from a help panel. The application then calls its own tutorial program.

2.5.4.3 Messages Sent to Help Manager

The application sends the following messages to Help Manager:

HM_CREATE_HELP_TABLE Specifies a new help table for the help instance.

HM_DISMISS_WINDOW Directs Help Manager to close the help window associated with the last active window.

HM_DISPLAY_HELP Directs Help Manager to display a specific help window.

HM_EXT_HELP Directs Help Manager to display the extended help panel for the active application window.

HM_HELP_CONTENTS Directs Help Manager to display the table of contents for the open help library.

HM_HELP_INDEX Directs Help Manager to display the index for the open help library.

HM_KEYS_HELP Directs Help Manager to display the help panel that contains information about the application keys.

HM_LOAD_HELP_TABLE Directs Help Manager to replace the existing help table with a help-table resource.

HM_REPLACE_HELP_FOR_HELP Directs Help Manager to display the application-defined help panel instead of the general help panel that is shipped with Help Manager.

HM_SET_ACTIVE_WINDOW Directs Help Manager to change the active window. Subsequent help messages are sent to the new active window and appear next to it.

HM_SET_HELP_LIBRARY_NAME Identifies the help library to the help instance.

HM_SET_HELP_WINDOW_TITLE Sets the title text of a help window.

HM_SET_SHOW_PANEL_ID Directs Help Manager to display, hide, or toggle the panel ID for each help panel displayed.

2.6 Combination-Box Controls

This section describes how to use combination-box controls to let the user choose and edit items from a list. Before reading this section, you should be familiar with entry-field controls, list-box controls, messages and message queues, and standard user-interface guidelines.

Combination-box controls, also called combo boxes, are a new feature of MS OS/2, version 1.2. They can be used in addition to entry-field controls, which are described in the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

2.6.1 About Combo Boxes

A combo box is two controls in one: an entry field and a list box. Combo boxes let the user enter data by typing in the entry field or by choosing from a list in the list box.

A combo box automatically manages the interaction between the entry field and the list box. For example, when the user chooses an item in the list box, the combo box displays the text for that item in the entry field. The user can then edit the text without affecting the item in the list box. When the user types a letter in the entry field, the combo box scrolls the list box contents so that items beginning with that letter become visible.

A combo box can have one of the following styles:

Style	Meaning
CBS_SIMPLE	A simple combo box. A simple combo box always displays its list box. The user can enter and edit text in the entry field or choose items from the list box.
CBS_DROPDOWN	A drop-down combo box. A simple drop-down combo box displays its list box only if the user clicks the drop-down icon at the right end of the entry field. It hides the list box when the user clicks the icon a second time. In a drop-down combo box, the user can enter and edit text in the entry field or choose items from the list box.
CBS_DROPDOWNLIST	A drop-down-list combo box is similar to the drop-down combo box, but the user can choose items only from the list box. The user cannot enter or edit text in the entry field.

For combo boxes that have the CBS_DROPDOWN or CBS_DROPDOWNLIST styles, an application can show the list by using the CBM_SHOWLIST message. An application can determine whether the list is already showing by using the CBM_ISLISTSHOWING message.

Applications can use any of the entry-field (EM_) and list-box (LM_) messages with combo boxes. Entry-field messages affect the entry field; list-box messages affect the list box. For example, an application can use the LM_INSERTITEM message to insert items into the list box. For more information on the entry-field and list-box messages, see the *Microsoft Operating System/2 Programmer's Reference, Volume 1* and *Volume 2*.

A combo box sends a variety of notification messages to its parent window. These notification messages are similar to the notification messages sent by entry-field and list-box controls. For example, the combo box sends a CBN_EFCHANGE notification message when the user changes text in the entry field and sends a CBN_LBSELECT when the user chooses an item in the list box.

2.6.2 Using Combo Boxes

You can create a combo box by using the **WinCreateWindow** function or by specifying a **COMBOBOX** statement in a dialog-window template in a resource file. When creating a combo box by using **WinCreateWindow**, you must specify the WC_COMBOBOX class, the predefined class for a combo box. If you do not specify a style, the default styles WS_GROUP, WS_TABSTOP, and WS_VISIBLE are used.

2.6.3 Summary

The following MS OS/2 styles and messages are used with combination-box controls.

2.6.3.1 Combo-Box Styles

The following style constants, specified when the combo box is created, determine its action and appearance:

CBS_SIMPLE Specifies a simple combo box made up of a list-box control and an entry-field control that are visible at all times.

CBS_DROPDOWN Specifies a drop-down combo box made up of an entry-field control and a button. When the user selects the button, a list-box control appears.

CBS_DROPDOWNLIST Similar to a drop-down combo box, but the user cannot enter or edit text in the entry field.

2.6.3.2 Messages Sent to a Combo Box

An application sends these messages to a combo box:

CBM_HILITE Sets drop-down button highlighting in a combo box.

CBM_ISLISTSHOWING Determines if a list box is visible in a combo box.

CBM_SHOWLIST Shows or hides the list box in a combo box.

2.6.3.3 Messages Sent by a Combo Box

Messages sent from a combo box to an owner window notify the owner of events in the combo box, such as when the user edits text. A combo box sends the following message to an owner window:

WM_CONTROL Sent to the owner window of the combo box when a user event occurs in the combo box. This message contains one of the following notification control codes, specifying what event occurred.

Code	Description
CBN_EFCHANGE	Indicates text in a combo-box entry field has changed.
CBN_EFSCROLL	Indicates a combo-box entry field is scrolled.
CBN_ENTER	Indicates a combo-box item is selected.
CBN_LBSCROLL	Indicates a combo-box list is scrolled.
CBN_LBSELECT	Indicates a combo-box list item is selected.
CBN_MEMERROR	Indicates the combo box cannot allocate sufficient memory.
CBN_SHOWLIST	Indicates a combo-box list has dropped down (is visible).

2.7 Multiple-Line Entry Fields

This section describes how to use multiple-line entry fields to let the user view and edit text in an application. Before reading this section, you should be familiar with entry-field controls, messages and message queues, and standard user-interface guidelines.

Multiple-line entry fields are a new feature of MS OS/2, version 1.2, and can be used in addition to entry-field controls, which are described in the *Microsoft Operating System/2 Programmer's Reference, Volume 1.*

2.7.1 About Multiple-Line Entry Fields

A multiple-line entry field (MLE) is a very sophisticated control window that users use to view and edit multiple lines of text. An MLE provides all the text-editing capability of a simple text editor, making these features readily available to applications.

You can create multiple-line entry fields by using the **WinCreateWindow** function or by specifying the MLE statement in a dialog-window template in a resource file.

2.7.1.1 Editing MLE Text

An MLE contains one or more lines of text. Each line consists of one or more characters and ends with one or more characters that represent the end of the line. The user inserts text by typing (when the MLE has the focus). The application can insert text at any time by using the MLM_INSERT message and specifying the text as a null-terminated string. The MLE inserts the new text at the cursor position or replaces the selected text.

The entry mode determines the action of the MLE when the user inserts text. The entry mode can be set to overstrike or insertion. The user sets it by pressing the INSERT key. When overstrike mode is enabled, at least one character is always selected. This means that the MLM_INSERT message always replaces at least one character. If insert mode is enabled, the MLM_INSERT message replaces only characters the user or the application has selected. Otherwise, the MLE makes room for the inserted characters by moving existing characters to the right at the cursor position.

The cursor position, identified by a flashing caret, is always specified as a character offset, relative to the beginning of text. The user sets the cursor position by moving the flashing caret using the mouse or the direction keys. An application can set the cursor position by using the MLM_SETSEL message. This message directs the MLE to move the flashing caret to a given character position.

The MLM_SETSEL message also sets the selection. The selection is one or more characters of text on which the MLE carries out an operation, such as deleting or copying. The user selects text by pressing the SHIFT key while moving the cursor. An application selects text by specifying the cursor position and anchor point using the MLM_SETSEL message. The selection is all text between the cursor position and the anchor point. If the cursor position and anchor point are equal, there is no selection. An application can retrieve the cursor position and/or anchor point by using the MLM_QUERYSEL message.

The user can delete characters, one at a time, by pressing the DELETE key or the BACKSPACE key. These keys delete the character to the left of the cursor. An application can delete one or more characters by using the MLM_DELETE message. This message directs the MLE to delete a specified number of characters, starting at the given position. This message does not change the cursor position. An application can delete selected text by using the MLM_CLEAR message.

An application can reverse the previous operation by using the MLM_UNDO message. This message directs the MLE to restore the entry field to its previous state. It is a quick way to fix users' editing mistakes.

But not all operations can be undone. The application can determine whether the previous operation can be undone by using the MLM_QUERYUNDO message. This message returns TRUE and an indication of the type of operation that can be undone. An application can prevent a subsequent MLM_UNDO message from changing the state of the MLE by using the MLM_RESETUNDO message.

2.7.1.2 Formatting MLE Text

An application can retrieve the number of lines of text in an MLE by using the MLM_QUERYLINECOUNT message. It can retrieve the number of characters in the MLE by using the MLM_QUERYTEXTLENGTH message. The amount of text and, subsequently, the number of lines to be entered in an MLE depend on the text limit. An application can set the text limit by using the MLM_SETTEXTLIMIT message and determine the current limit by using the MLM_QUERYTEXTLIMIT message. The user cannot set the limit. If the user types to the text limit, the MLE beeps and ignores subsequent characters. If the application attempts to add text beyond the limit, the MLE truncates the text.

An application can control the length of each line in an MLE by enabling word-wrapping. When word-wrapping is enabled, the MLE automatically breaks any line that is longer than the MLE is wide. An application can set word-wrapping by using the MLM_SETWRAP message, and it can determine whether the MLE is wrapping text by using the MLM_QUERYWRAP message. Unless the MLS_WORDWRAP style is specified when the MLE is created, word-wrapping is initially disabled.

An application can set tab stops for an MLE by using the MLM_SETTABSTOP message. Tab stops specify the maximum width of tab character. When the user or an application inserts a tab character, the MLE expands the character so that it fills the space between cursor position and the next tab stop. The MLM_SETTABSTOP message actually sets the distance (specified in pels) between tab stops, and the MLE provides as many tab stops as needed, no matter how long the line gets. An application can retrieve the distance between tab stops by using the MLM_QUERYTABSTOP message.

An application can use the MLM_SETFORMATRECT message to set the format rectangle. The format rectangle is used to set the horizontal and/or vertical limits for text. The MLE sends a notification message to the parent window of the MLE if text exceeds the limit. An application typically uses the format rectangle to provide its own word-wrapping or other special text processing. An application can retrieve the current formatting rectangle by using the MLM_QUERYFORMATRECT message.

An application can prevent the user from entering text in the entry field by using the MLM_SETREADONLY message. The MLM_QUERYREADONLY message specifies whether the MLE is read-only. An application can also set the MLE to read-only by specifying the MLS_READONLY style when creating the MLE.

An application can set the colors and font for an MLE by using the MLM_SETTEXTCOLOR, MLM_SETBACKCOLOR, and MLM_SETFONT messages. These messages affect all text in the MLE; an MLE cannot contain a mixture of fonts and colors. An application can retrieve the current values for the color and the font by using the MLM_QUERYTEXTCOLOR, MLM_QUERYBACKCOLOR, and MLM_QUERYFONT messages.

2.7.1.3 Importing and Exporting MLE Text

An application can copy text to and from an MLE by importing and exporting. Importing using the MLM_IMPORT message copies text from a buffer to the MLE. Exporting using the MLM_EXPORT message copies text from the MLE to a buffer. The application uses the MLM_SETIMPORTEXPORT message to set the import and export buffers. To import, the application must fill the buffer with the text to copy to the MLE. To export, the MLE copies the specified text to the buffer.

An application can import and export text in a variety of formats. The text format identifies which characters are used for the end-of-line characters and is set using the MLM_FORMAT message. An MLE can have the following text formats:

Type	Format
MLFIE_CFTEXT	Exported lines end with a carriage-return/newline character pair (0x0D, 0x0A). Imported lines must end with a newline character, a carriage-return/newline character pair, or a newline/carriage-return character pair.
MLFIE_NOTRANS	Imported and exported lines end with a newline character (0x0A).
MLFIE_WINFMT	For exported lines, the carriage-return/newline character pair marks a hard line break (a break entered by the user), and two carriage-return characters and a newline character (0x0D, 0x0D, 0x0A) mark a soft line break (a break inserted during word-wrapping, not entered by the user). For imported lines, soft line break characters are ignored.

The text format can affect the number of characters in a selection. To ensure the export buffer is large enough to hold exported text, an application can send the MLM_QUERYFORMATLINELENGTH message and the MLM_QUERYFORMATTEXTLENGTH message to determine the number of bytes in text to be exported.

Each time an application inserts text in an MLE, the MLE automatically refreshes the display by drawing the new text. When an application copies large amounts of text to an MLE, refreshing can be quite time-consuming, so applications should disable the automatic refresh setting in such cases. An application can disable this setting by sending the MLM_DISABLEREFRESH message. After copying all the text, the application can restore the refresh by sending the MLM_ENABLEREFRESH message.

2.7.1.4 Copying and Pasting MLE Text

The user can cut, copy, and paste text in an MLE by using the CTRL+DELETE, SHIFT+DELETE, and SHIFT+INSERT keys. An application can cut, copy, and paste text by using the MLM_CUT, MLM_COPY, and MLM_PASTE messages. The MLM_CUT and MLM_COPY messages direct the MLE to copy the selected text to the clipboard. The MLM_CUT message also deletes the text (MLM_COPY does not). The MLM_PASTE message directs the MLE to copy the text on the clipboard to the current position in the MLE, replacing any existing text with the copied text. An application can delete the selected text without copying it to the clipboard by using the MLM_CLEAR message.

An application can also copy the selected text from an MLE to a buffer by using the MLM_QUERYSELTEXT message. This message does not affect the contents of the clipboard.

2.7.1.5 Searching and Replacing MLE Text

An application can search for a specified string within MLE text by using the MLM_SEARCH message. This message directs the MLE to search for the string. If the string is found, the MLE returns TRUE. The cursor does not move to the string unless the message specifies the MLFSEARCH_SELECTMATCH option.

An application can also use the MLM_SEARCH message to replace one string with another. If the MLFSEARCH_CHANGEALL option is specified, the MLE replaces all occurrences of the search string with the replacement string. Both the search string and the replacement string must be given in a **MLE_SEARCHDATA** structure passed with the message.

2.7.1.6 MLE Notification Codes

An MLE sends notification codes to its parent window whenever certain events occur, for example, when the user or the application tries to insert too much text or when the user uses the scroll bars. The parent window uses the notification codes to carry out custom operations for the MLE or to respond to errors. Notification codes that are closely related to MLE messages are described here.

The MLE sends the MLN_HSCROLL or MLN_VSCROLL notification codes when the user uses the scroll bars so the application can monitor the visible contents of the MLE. The application can also monitor the contents of an MLE by using the MLM_QUERYFIRSTCHAR message. This message identifies the character in the upper-left corner of the MLE (by specifying its offset). This represents the first MLE character that is visible to the user. An application can move a specified character to the upper-left corner of an MLE by using the MLM_SETFIRSTCHAR message as an alternative way of scrolling the contents of an MLE.

The MLE sends an MLN_CHANGE notification code when the user changes the text in some way. This code is especially useful when the MLE is in a dialog box because it can determine whether the dialog procedure should process the contents of the MLE. The MLM_QUERYCHANGED message also can determine whether the user has made changes. The MLM_SETCHANGED message causes the MLE to send a notification code, regardless of whether the user has changed anything; this code can also be used to hide a change made by a user.

2.7.1.7 MLE Styles

MLE styles can be specified by using the **WinCreateWindow** function or the MLE statement in a resource file. Styles can be combined by using the OR operator. Applications can specify a combination of the following styles for an MLE:

Style	Meaning
MLS_BORDER	Draws a border around the MLE.
MLS_HSCROLL	Adds a horizontal scroll bar to the MLE. The scroll bar is enabled when any line exceeds the width of the MLE.
MLS_IGNORETAB	Directs the MLE to ignore the TAB key.
MLS_READONLY	Prevents the MLE from accepting text from the user. This style is useful for displaying lengthy static text in windows or dialog boxes.
MLS_VSCROLL	Adds a vertical scroll bar to the MLE. The scroll bar is enabled when the number of lines exceeds the height of the MLE.
MLS_WORDWRAP	Prevents lines that are longer than the width of the MLE. The MLE automatically breaks the line at a convenient place.

2.7.2 Using Multiple-Line Entry Fields

You can create an MLE by using the **WinCreateWindow** function or by specifying the **MLE** statement in a dialog-window template in a resource file. The following example shows how to create an MLE using **WinCreateWindow**:

```
HWND hwndParent;    /* parent-window handle  */
HWND hwndMLE;       /* MLE handle            */

hwndMLE = WinCreateWindow(hwndParent,
              WC_MLE,
              "Test",
              MLS_BORDER | WS_VISIBLE,
              100, 100, 100, 100,
              hwndParent,
              HWND_TOP,
              2, NULL, NULL);
```

An MLE has the WC_MLE window class. As with other controls created using the **WinCreateWindow** function, the WS_VISIBLE style must be set to display the window immediately.

It is more common to create an MLE by using an MLE statement in a dialog-window template in a resource file, as shown in the following example:

```
MLE "", 101, 110, 10, 50, 100
```

The predefined class for an MLE is WC_MLE. If you do not specify a style, the default styles MLS_BORDER, WS_GROUP, and WS_TABSTOP are used.

2.7.3 Summary

The following MS OS/2 styles and messages are used with multiple-line entry fields.

2.7.3.1 MLE Styles

The following style constants, specified when the MLE is created, determine its action and appearance:

MLS_BORDER   Places a thin border around the MLE.

MLS_HSCROLL   Adds a horizontal scroll bar to the MLE.

MLS_IGNORETAB   Prevents the TAB key from functioning in the MLE.

MLS_READONLY   Makes the MLE text read-only. The user cannot enter or edit text in the MLE.

MLS_VSCROLL   Adds a vertical scroll bar to the MLE.

MLS_WORDWRAP   Automatically moves words that do not fit at the end of a line to the next line.

2.7.3.2 Messages Sent to an MLE

An application sends the following messages to an MLE:

MLM_CHARFROMLINE   Returns the offset to a line.

MLM_CLEAR   Clears selected text in an MLE.

MLM_COPY   Copies selected text from an MLE to the clipboard.

MLM_CUT   Cuts selected text from an MLE to the clipboard.

MLM_DELETE   Deletes text from an MLE.

MLM_DISABLEREFRESH   Disables refresh for an MLE.

MLM_ENABLEREFRESH   Enables screen refresh for an MLE.

MLM_EXPORT   Exports text from an MLE.

MLM_FORMAT   Sets format for MLE import/export.

MLM_IMPORT   Imports text into an MLE.

MLM_INSERT   Inserts text into an MLE.

MLM_LINEFROMCHAR   Determines the line number of an MLE character.

MLM_PASTE   Copies the clipboard contents to an MLE.

MLM_QUERYBACKCOLOR   Retrieves the background color of an MLE.

MLM_QUERYCHANGED Determines if text in an MLE has changed.

MLM_QUERYFIRSTCHAR Retrieves the offset of the first visible character.

MLM_QUERYFONT Retrieves current MLE font information.

MLM_QUERYFORMATLINELENGTH Retrieves the formatted MLE line length.

MLM_QUERYFORMATRECT Retrieves the dimensions and mode of an MLE.

MLM_QUERYFORMATTEXTLENGTH Retrieves the length of formatted MLE text.

MLM_QUERYIMPORTEXPORT Retrieves values for the import/export buffer.

MLM_QUERYLINECOUNT Retrieves the number of lines in an MLE.

MLM_QUERYLINELENGTH Retrieves the length of an MLE line.

MLM_QUERYREADONLY Determines MLE read-only mode.

MLM_QUERYSEL Retrieves the selection position in an MLE.

MLM_QUERYSELTEXT Retrieves selected text from an MLE.

MLM_QUERYTABSTOP Retrieves the size of an MLE tab-stop.

MLM_QUERYTEXTCOLOR Retrieves MLE text-color information.

MLM_QUERYTEXTLENGTH Retrieves the length of MLE text.

MLM_QUERYTEXTLIMIT Retrieves the text limit of an MLE.

MLM_QUERYUNDO Determines if an MLE can undo an operation.

MLM_QUERYWRAP Retrieves the state of word-wrap in an MLE.

MLM_RESETUNDO Resets (clears) the MLE undo flag.

MLM_SEARCH Searches an MLE.

MLM_SETBACKCOLOR Sets the background color of an MLE.

MLM_SETCHANGED Sets the MLE changed flag.

MLM_SETFIRSTCHAR Sets the first visible character.

MLM_SETFONT Sets MLE font information.

MLM_SETFORMATRECT Sets the format rectangle and mode of an MLE.

MLM_SETIMPORTEXPORT Sets the MLE import/export buffer.

MLM_SETREADONLY Sets/clears the MLE read-only state.

MLM_SETSEL Selects text within an MLE.

MLM_SETTABSTOP Sets the size of an MLE tab-stop.

MLM_SETTEXTCOLOR Sets the text color of an MLE.

MLM_SETTEXTLIMIT Sets the text limit for an MLE.

MLM_SETWRAP Sets/resets MLE word-wrap.

MLM_UNDO Undoes an MLE operation.

2.7.3.3 Messages Sent from an MLE

Messages sent from an MLE to an owner window notify the owner of events in the MLE, such as when the user edits text. An MLE sends the following message to an owner window:

WM_CONTROL Sent to the owner window of the MLE when a user event occurs in the MLE. This message contains one of the following notification control codes, specifying what event occurred.

Code	Description
MLN_CHANGE	Indicates that text in an MLE has changed.
MLN_CLPBDFAIL	Indicates that a clipboard operation failed.
MLN_HSCROLL	Indicates a horizontal MLE scroll event.
MLN_KILLFOCUS	Indicates an MLE has lost the input focus.
MLN_MARGIN	Indicates the mouse has moved over an MLE margin.
MLN_MEMERROR	Indicates insufficient memory available for an MLE.
MLN_OVERFLOW	Indicates the MLE operation caused an overflow.
MLN_PIXHORZOVERFLOW	Indicates an MLE horizontal overflow.
MLN_PIXVERTOVERFLOW	Indicates an MLE vertical overflow.
MLN_SEARCHPAUSE	Determines the status of a search initiated by an MLM_SEARCH message.
MLN_SETFOCUS	Indicates the MLE receives the input focus.
MLN_TEXTOVERFLOW	Indicates an MLE text-limit overflow.
MLN_UNDOOVERFLOW	Indicates a text change cannot be undone.
MLN_VSCROLL	Indicates an MLE vertical scroll event.

Chapter
3

Functions and Messages Directory

3.1 Introduction 53
3.1.1 Function Groups 53
3.1.2 Message Groups 54
3.2 Directory 55
3.3 Functions and Messages 56

3.1 Introduction

This chapter describes MS OS/2 system functions and messages that are new or modified for MS OS/2, version 1.2. These functions provide features, such as multiple-line entry fields, extended attributes for disk files, and application help. The functions and messages represent distinct functional groups.

3.1.1 Function Groups

Programs use the function groups described in the following list to carry out specific tasks.

Function group	Usage
Dev	Use the Presentation Manager device (**Dev**) functions to open and control Presentation Manager device drivers. These functions let you create device contexts that you can associate with a presentation space and use with the **Gpi** functions to carry out device-independent graphics for displays, printers, and plotters.
Dos	Use the disk operating system (**Dos**) functions in full-screen and Presentation Manager sessions to read from and write to disk files, to allocate memory, to start threads and processes, to communicate with other processes, and to access your computer's devices directly. Most functions in this group can be used in Presentation Manager applications.
Gpi	Use the graphics programming interface (**Gpi**) functions to create graphics output for displays, printers, and other output devices. The **Gpi** functions give you a full range of graphics primitives, from lines to complex curves to bitmaps. You choose the attributes for the primitives, such as color, line width, and pattern, and then draw lines, text, and shapes. The retained-graphics capability lets you save the drawing in segments and build complex pictures by drawing a chain of segments.
Kbd	Use the keyboard (**Kbd**) functions in full-screen sessions to read keystrokes from the keyboard, to manage multiple logical keyboards, and to change code pages and translation tables. Because the Presentation Manager session provides its own keyboard support, **Kbd** functions are not needed in Presentation Manager applications.

Function group	Usage
Mou	Use the mouse (**Mou**) functions in full-screen sessions to read mouse input from the mouse-event queue, to set the mouse-pointer shape, and to manage the mouse for all processes in a session. As with the keyboard, the Presentation Manager session provides its own mouse support, so **Mou** functions are not needed in Presentation Manager applications.
Pic	Use the picture-file (**Pic**) functions when working with picture files, typically either metafiles or interchange files.
Prf	Use the Profile Manager (**Prf**) functions to open and modify the MS OS/2 initialization files, *os2.ini* and *os2sys.ini*. The **Prf** functions let you store application information in the initialization files, making that information available to other applications or to the application itself after it has been stopped and restarted.
Vio	Use the video input-and-output (**Vio**) functions in full-screen sessions to write characters and character attributes to the screen, to create pop-up windows for messages, to change the video modes, and to access physical video memory. **Vio** functions can also be used in advanced video-input-and-output (AVIO) applications for the Presentation Manager session to write characters and character attributes in a window. Most Presentation Manager applications, however, use the graphics programming interface (Gpi) to write text in a window.
Win	Use the window-manager (**Win**) functions to create and manage windows. Presentation Manager applications use windows as the main interface with the user. The **Win** functions let you create menus, scroll bars, and dialog windows that let the user choose commands and supply input. Your application receives all mouse and keyboard input as messages from the message queue. The **Win** functions let you retrieve messages from the queue and dispatch them to the window the input is intended for.

3.1.2 Message Groups

MS OS/2 uses system-defined messages that control the operation of applications. The messages are divided into groups according to the various types of windows that can interpret and process the messages. Applications use the message groups described in the following list to carry out specific tasks.

Message group	Usage
Combination box	Use the combo-box control messages (CBM_) to control combination boxes.
Entry field	Use the entry-field control messages (EM_) to control entry fields.
Help Manager	Use the Help Manager messages (HM_) to direct Help Manager for your applications.
Multiple-line entry field	Use the multiple-line entry-field messages (MLM_) to control multiple-line entry fields.
Menus	Use the menu messages (MM_) to control menus and menu items.
Scroll bar	Use the scroll-bar messages (SBM_) to control scroll bars and sliders.
Title bar	Use the title-bar messages (TBM_) to control title bars.
General	Use the general window messages (WM_) to control the operation of windows of any window class. For most general window messages, the system sends the message to the window procedure of the given window. These messages can represent input from the keyboard, mouse, or timer. Some messages are requests from the system to the window procedure for information, or they are actions to be taken. Other messages contain information that the window procedure can use or save for processing later. MS OS/2 uses general window messages when creating, destroying, moving, sizing, and activating windows. It also uses these messages for all input to the window, whether the input is from devices, such as the keyboard and mouse, or through other windows, such as dialogs and menus.

3.2 Directory

The remainder of this chapter is a directory that gives complete syntax, purpose, and parameter descriptions for MS OS/2, version 1.2, functions and messages. The types, macros, and structures used by a function are given with the function, and they are described more fully in Chapter 4, "Types, Macros, Structures." You will notice the word *New*, *Change*, or *Correction* on the right side of the line that contains the function or message name. This heading tells you whether that particular function or message is new to MS OS/2, version 1.2; changed, or updated, from MS OS/2, version 1.1; or contains a correction to an error that appeared in MS OS/2, version 1.1 documentation.

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

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

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

3.3 Functions and Messages

The following list, in alphabetical order, details the new, changed, and corrected functions and messages for MS OS/2, version 1.2.

■ CBM_HILITE New

```
CBM_HILITE
mp1 = MPFROMSHORT((USHORT) fHilite);     /* highlight flag          */
mp2 = 0L;                                /* not used, must be zero  */
```

An application sends a CBM_HILITE message to set the highlighting state of the drop-down list button in a combination box that was created with the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

Parameters

fHilite Low word of *mp1*. Specifies whether to highlight or remove highlighting from the drop-down list button. If this parameter is TRUE, the system highlights the button; if it is FALSE, the system removes the highlighting.

Return Value

The return value is TRUE if the state of the drop-down list button changes or FALSE if it does not.

■ CBM_ISLISTSHOWING New

```
CBM_ISLISTSHOWING
mp1 = 0L;     /* not used, must be zero */
mp2 = 0L;     /* not used, must be zero */
```

An application sends a CBM_ISLISTSHOWING message to determine whether the list box in a combination box is currently displayed.

Parameters

This message does not use any parameters.

Return Value

The return value is TRUE if the list box is displayed or FALSE if it is not.

See Also

CBM_SHOWLIST

■ CBM_SHOWLIST New

```
CBM_SHOWLIST
mp1 = MPFROMSHORT((USHORT) fShow);     /* show flag              */
mp2 = 0L;                              /* not used, must be zero */
```

An application sends a CBM_SHOWLIST message to show or hide the list box in a combination box.

Parameters

fShow Low word of *mp1*. Specifies whether to show or hide the list box. If this parameter is TRUE, the list box is shown; otherwise, it is hidden.

Return Value

The return value is TRUE if the state of the list box changes or FALSE if it does not change.

See Also

CBM_ISLISTSHOWING

■ CBN_EFCHANGE New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID */
usNotifyCode = CBN_EFCHANGE;
```

The CBN_EFCHANGE notification message is sent when the text in a combination-box entry field changes.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_EFCHANGE.

See Also

WM_CONTROL

■ CBN_EFSCROLL New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID */
usNotifyCode = CBN_EFSCROLL;
```

The CBN_EFSCROLL notification message is sent when a combination-box entry field is scrolled.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_EFSCROLL.

Return Value

An application should return zero if it processes this message.

See Also

WM_CONTROL

■ CBN_ENTER New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID */
usNotifyCode = CBN_ENTER;
```

The CBN_ENTER notification message is sent when the user presses the ENTER key or double-clicks a list item in a combination box.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_ENTER.

Return Value

An application should return zero if it processes this message.

See Also

WM_CONTROL

■ CBN_LBSCROLL New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* control-window ID */
usNotifyCode = CBN_LBSCROLL;
```

The CBN_LBSCROLL notification message is sent when a combination-box list is scrolled.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_LBSCROLL.

Return Value An application should return zero if it processes this message.

See Also WM_CONTROL

■ CBN_LBSELECT New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* control-window ID */
usNotifyCode = CBN_LBSELECT;
```

The CBN_LBSELECT notification message is sent when a combination-box list item is selected.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_LBSELECT.

Return Value An application should return zero if it processes this message.

See Also WM_CONTROL

■ CBN_MEMERROR New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* control-window ID */
usNotifyCode = CBN_MEMERROR;
```

The CBN_MEMERROR notification message is sent when a combination-box cannot allocate the amount of memory necessary.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_MEMERROR.

Return Value An application should return zero if it processes this message.

See Also WM_CONTROL

■ CBN_SHOWLIST New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID */
usNotifyCode = CBN_SHOWLIST;
```

The CBN_SHOWLIST notification message is sent when the combination-box list is shown (dropped down).

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to CBN_SHOWLIST.

Return Value

An application should return zero if it processes this message.

See Also

WM_CONTROL

■ DevEscape Change

```
LONG DevEscape(hdc, cmdCode, cbInData, pbInData, pcbOutData, pbOutData)
HDC hdc;                /* device-context handle                        */
LONG cmdCode;           /* escape function to perform                   */
LONG cbInData;          /* size of input buffer                         */
PBYTE pbInData;         /* pointer to input buffer                      */
PLONG pcbOutData;       /* pointer to buffer for bytes in output buffer */
PBYTE pbOutData;        /* pointer to output-data buffer                */
```

The **DevEscape** function allows applications to access facilities of a device not otherwise available through the API. Because calls to escape functions are generally sent to the device driver, the device driver must be able to use them.

Parameters

hdc Identifies the device context.

cmdCode Specifies the escape function to perform. The following escape functions are currently defined:

DEVESC_ABORTDOC
DEVESC_BREAK_EXTRA
DEVESC_CHAR_EXTRA
DEVESC_DRAFTMODE
DEVESC_ENDDOC
DEVESC_FLUSHOUTPUT
DEVESC_GETSCALINGFACTOR
DEVESC_NEWFRAME
DEVESC_NEXTBAND
DEVESC_QUERYESCSUPPORT
DEVESC_QUERYVIOCELLSIZES
DEVESC_RAWDATA
DEVESC_STARTDOC

Devices can define additional escape functions by using other *cmdCode* values in the following ranges:

Range	Meaning
32768–40959	Not stored in a metafile and not recorded.
40960–49151	Stored in a metafile only.
49152–57343	Stored in a metafile and recorded.
57344–65535	Recorded only.

cbInData Specifies the number of bytes of data in the buffer pointed to by the *pbInData* parameter.

pbInData Points to the buffer that contains the input data required for the escape function.

pcbOutData Points to the buffer that receives the number of bytes of data in the buffer pointed by the *pbOutData* parameter. If data is returned in the *pbOutData* parameter, *pcbOutData* is updated to the number of bytes of data returned.

pbOutData Points to the buffer that receives the output from the escape function. If this parameter is NULL, no data is returned.

Return Value

The return value is DEV_OK if the function is successful, DEVESC_ERROR if an error occurs, or DEVESC_NOTIMPLEMENTED if the escape function is not implemented for the specified code.

Errors

You can use the **WinGetLastError** function to retrieve the error value, which may be one of the following values:

PMERR_ESC_CODE_NOT_SUPPORTED
PMERR_INV_ESCAPE_DATA
PMERR_INV_HDC
PMERR_INV_LENGTH_OR_COUNT

Comments

The standard escape functions and the corresponding **DevEscape** parameters are listed in the following paragraphs.

The DEVESC_BREAK_EXTRA escape defines extra width to add to the break character when that character is transmitted to the device specified by the *hdc* parameter. The extra width is used in aligning text. The **GpiQueryFonts** function can be used to determine the break character used in a specific font.

For DEVESC_BREAK_EXTRA, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Specifies the number of bytes pointed to by the *pbInData* parameter. This parameter must be either zero (for no extra spacing) or 4 (for extra spacing).
pbInData	Points to the fixed-point number (**FIXED**) that specifies the amount of extra width (in world coordinate units) to add to the break character.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be zero.

Extra spacing is initialized to zero whenever a display context is opened. Any change made to the extra spacing remains in effect until either the display context is closed or a new change to the extra spacing is made.

The DEVESC_CHAR_EXTRA escape defines extra width to add to all characters when they are transmitted to the device specified by the *hdc* parameter. The extra width is used in aligning text.

For DEVESC_CHAR_EXTRA, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Specifies the number of bytes pointed to by the *pbInData* parameter. This parameter must be either zero (for no extra spacing) or 4 (for extra spacing).
pbInData	Points to the fixed-point number (**FIXED**) that specifies the amount of extra width to be added.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be zero.

Extra spacing is initialized to zero whenever a display context is opened. Any change made to the extra spacing remains in effect until either the display context is closed or a new change to the extra spacing is made.

The extra width added to the break character is the sum of the break-extra and character-extra amounts. Providing a width vector to **GpiCharStringPos** or **GpiQueryCharStringPosAt** operates in addition to the extra spacing feature. Extra spacing does not override kerning; extra spacing adjustments and kerning adjustments simply sum.

Text drawn in a path is not affected by the extra spacing. This means that outlined text and text used for a clipping region are displayed as if the extra spacing fields were set to zero.

The DEVESC_QUERYESCSUPPORT escape determines whether the device driver has implemented a particular escape. The return value gives the result. This escape is not stored in a metafile or recorded.

For DEVESC_QUERYESCSUPPORT, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Specifies the number of bytes pointed to by the *pbInData* parameter.
pbInData	Specifies the escape-code value of the escape function to be checked.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be zero.

The DEVESC_QUERYVIOCELLSIZES escape returns the cell sizes supported by the device identified by the *hdc* parameter.

For DEVESC_QUERYVIOCELLSIZES, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.
pcbOutData	Points to the number of bytes of data pointed to by the *pbOutData* parameter. Upon return, this parameter contains to the number of bytes returned.

pbOutData	Points to the buffer that receives the output from this escape function. The output is returned in a **VIOSIZECOUNT** structure and an array of **VIOFONTCELLSIZE** structures. These structures have the following forms:
	`typedef struct _VIOSIZECOUNT {`
	`   LONG maxcount;`
	`   LONG count;`
	`   } VIOSIZECOUNT;`
	
	`typedef struct _VIOFONTCELLSIZE {`
	`   LONG cx;`
	`   LONG cy;`
	`   } VIOFONTCELLSIZE;`
	
	The number of **VIOFONTCELLSIZE** structures returned is dependent on the value of the **count** field of the **VIOSIZECOUNT** structure.
	
	For a full description, see Chapter 4, "Types, Macros, Structures."

The DEVESC_GETSCALINGFACTOR escape returns the scaling factors for the *x* and *y* axes of a printing device. For each scaling factor, an exponent of two is put in the *pbOutData* parameter. For example, the value 3 is used if the scaling factor is 8. Scaling factors are used by devices that cannot support graphics at the same resolution as the device resolution.

For DEVESC_GETSCALINGFACTOR, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.
pcbOutData	Points to the number of bytes of data pointed to by the *pbOutData* parameter. Upon return, this parameter contains the number of bytes returned.
pbOutData	Points to the buffer that receives the output from this escape. A structure is returned that specifies the scaling factors for the *x* and *y* axes.

The DEVESC_STARTDOC escape indicates the start of a new print job. All subsequent output to the device context, up to the next DEVESC_ENDDOC escape, is spooled under the same job.

For DEVESC_STARTDOC, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Specifies the number of bytes pointed to by the *pbInData* parameter.
pbInData	Points to the null-terminated string that specifies the name of the document.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be NULL.

The DEVESC_ENDDOC escape ends a print job started by the DEVESC_STARTDOC escape.

For DEVESC_ENDDOC, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.
pcbOutData	Points to the buffer that specifies the number of characters in the string pointed to by the *pbOutData* parameter. This parameter should be NULL if the number of characters is zero.
pbOutData	Points to the unsigned 16-bit integer that specifies the job identifier if a spooler print job was created.

The DEVESC_NEXTBAND escape allows an application to signal that it has finished writing to a "band," or rectangle. The coordinates of the next band are returned. This escape is used by applications that perform handle banding ("for-printing") themselves.

For DEVESC_NEXTBAND, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.
pcbOutData	Points to the number of bytes of data pointed to by the *pbOutData* parameter. Upon return, this parameter contains the number of bytes returned.
pbOutData	Points to the address of the buffer that receives the output from this escape. A structure is returned that specifies the device coordinates of the next band.

The DEVESC_ABORTDOC escape stops the current job, erasing everything written by the application to the device since the last call to the DEVESC_ENDDOC escape function.

For DEVESC_ABORTDOC, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be NULL.

The DEVESC_NEWFRAME escape allows an application to signal when it has finished writing to a page. This escape is typically used with a printer device to advance to a new page. Using this escape is similar to processing the **GpiErase** function for a screen device context.

For DEVESC_NEWFRAME, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be NULL.

The DEVESC_DRAFTMODE escape turns draft mode on or off. Turning draft mode on instructs the device driver to print faster and, if necessary, with lower quality. You can change the draft mode only at page boundaries—for example, after a DEVESC_NEWFRAME escape.

For DEVESC_DRAFTMODE, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Specifies the number of bytes pointed to by the *pbInData* parameter.
pbInData	Points to the signed 16-bit integer that specifies the draft mode. This value is 1 if draft mode is on and zero if draft mode is off.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be NULL.

The DEVESC_FLUSHOUTPUT escape removes any output from the device buffer.

For DEVESC_FLUSHOUTPUT, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Not used; can be zero.
pbInData	Not used; can be NULL.

Parameter	Description
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be NULL.

The DEVESC_RAWDATA escape allows an application to send data directly to a device driver. For example, in the case of a printer device driver, the data could be a printer data stream.

If raw data is mixed with other data—for example, **Gpi** data—being sent to the same page of a device context, the results are unpredictable and depend upon the action taken by the Presentation Manager device driver, which, in this case, might ignore the GPI data completely. In general, you should send raw data either to a separate page, using the DEVESC_NEWFRAME escape to obtain a new page, or to a separate document, using the DEVESC_STARTDOC and DEVESC_ENDDOC escapes to create a new document.

For DEVESC_RAWDATA, the **DevEscape** parameters contain the following information:

Parameter	Description
cbInData	Specifies the number of bytes pointed to by the *pbInData* parameter.
pbInData	Points to the raw data.
pcbOutData	Not used; can be NULL.
pbOutData	Not used; can be NULL.

See Also **GpiErase**

Changes The escape functions DEVESC_BREAK_EXTRA, DEVESC_CHAR_EXTRA, and DEVESC_QUERYVIOCELLSIZES have been added.

The DEVESC_STARTDOC and DEVESC_ENDDOC escapes indicate the start and end of a print job.

DevPostDeviceModes — Correction

```
LONG DevPostDeviceModes( hab, pbDriverData, pszDriverName, achDeviceName, pszName, flOptions)
HAB hab;                    /* anchor-block handle                    */
PDRIVDATA pbDriverData;     /* pointer to buffer for data             */
PSZ pszDriverName;          /* pointer to string for driver name      */
PSZ achDeviceName;          /* pointer to device name                 */
PSZ pszName;                /* pointer to string for output device name */
ULONG flOptions;            /* specifies various options              */
```

The **DevPostDeviceModes** function causes a device driver to post a dialog box so the user can set options for the device (resolution, font cartridges, and so on).

The application can call this function first with a NULL data pointer to find how much storage is needed for the data buffer. It then calls the function a second time to have the buffer filled with data. You can then pass the returned data to the **DevOpenDC** function as the buffer data pointed to by the *pbDriverData* parameter.

Parameters

hab Identifies the anchor block.

pbDriverData Points to the data buffer that receives device data defined by the driver. If this parameter is NULL, the function returns the required buffer size. The format of the data is the same as for the *pbData* parameter of the **DevOpenDC** function.

pszDriverName Points to the null-terminated string that contains the name of the device driver.

achDeviceName Points to the null-terminated string that identifies the particular device (for example, its model number). This string must not exceed 32 bytes. Valid names are defined by device drivers.

pszName Points to the null-terminated string that contains the printer name.

flOptions Specifies whether the function should display a dialog box that allows the user to change job properties, display two dialog boxes that allow the user to change job and printer properties, or simply return the current job properties. This parameter can be one of the following values:

Value	Meaning
DPDMF_POSTJOBPROP	Display a dialog box that allows the user to change job properties. The default values for this dialog box are taken from the PM_SPOOLER_DD section of the *os2.ini* file if the *pszName* parameter specifies a logical address. If *pszName* is NULL, the default values are taken from the *pbDriverData* parameter.
DPDMF_CHANGEPROP	Display two dialog boxes. The first dialog box allows the user to change job properties; the second allows the user to change printer properties. The default values for these dialog boxes are taken from the PM_SPOOLER_DD section of the *os2.ini* file. The function returns the new values in the *pbDriverData* parameter. The *pszName* parameter cannot be NULL when this option is selected.
DPDMF_QUERYJOBPROP	Return the current job properties.

Return Value

The return value, if the *pbDriverData* parameter is NULL, is the size (in bytes) required for the data buffer, DPDM_NONE if there are no settable options, or DPDM_ERROR if an error occurs.

The return value, if *pbDriverData* is not NULL, is DEV_OK if the function is successful, DPDM_NONE if there is no device mode, or DPDM_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_DEVICE_NAME
PMERR_INV_DRIVER_DATA
PMERR_INV_DRIVER_NAME
PMERR_INV_LOGICAL_ADDRESS

See Also

DevOpenDC

Corrections

The sixth parameter (*flOptions*) was omitted in the previous description of the function.

DevQueryCaps — Correction

```
BOOL DevQueryCaps( hdc, lStartitem, cItems, alItems )
HDC hdc;              /* device-context handle          */
LONG lStartitem;      /* first item to be returned      */
LONG cItems;          /* number of items to be returned */
PLONG alItems;        /* array for device characteristics */
```

The **DevQueryCaps** function queries the characteristics of the specified device.

Parameters

hdc Identifies the device context.

lStartitem Specifies the first item of information to be returned in the array.

cItems Specifies the number of items to be returned in the array.

alItems Points to an array of device characteristics, starting with the item specified by the *lStartitem* parameter.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HDC
PMERR_INV_LENGTH_OR_COUNT
PMERR_INV_QUERY_ELEMENT_NO

Comments

The following are possible values for the *alItems* parameter:

CAPS_FAMILY Specifies the device type. These values are the same as the values for the *type* parameter in the **DevOpenDC** function.

CAPS_IO_CAPS Specifies the device input/output capability. The possible values are as follows:

Value	Meaning
CAPS_IO_DUMMY	Dummy device
CAPS_IO_SUPPORTS_OP	Output
CAPS_IO_SUPPORTS_IP	Input
CAPS_IO_SUPPORTS_IO	Output and input

CAPS_TECHNOLOGY Specifies the technology. The possible values are as follows:

Value	Meaning
CAPS_TECH_UNKNOWN	Unknown (for example, metafile)
CAPS_TECH_VECTOR_PLOTTER	Vector plotter
CAPS_TECH_RASTER_DISPLAY	Raster display
CAPS_TECH_RASTER_PRINTER	Raster printer
CAPS_TECH_RASTER_CAMERA	Raster camera
CAPS_TECH_POSTSCRIPT	PostScript printer

CAPS_DRIVER_VERSION Specifies the device-driver version number.

CAPS_HEIGHT Specifies the media depth (for a full-screen maximized window on a display) in pels. (For a plotter, a pel is defined as the smallest possible displacement of the pen and can be smaller than a pen width.)

CAPS_WIDTH Specifies the media width (for a full-screen, maximized window for displays) in pels.

CAPS_HEIGHT_IN_CHARS Specifies the media depth (for a full-screen, maximized window for displays) in character rows, for **Vio** calls only.

CAPS_WIDTH_IN_CHARS Specifies the media width (for a full-screen, maximized window for displays) in character columns, for **Vio** calls only.

CAPS_VERTICAL_RESOLUTION Specifies the vertical resolution (in pels per meter) of the device.

CAPS_HORIZONTAL_RESOLUTION Specifies the horizontal resolution (in pels per meter) of the device.

CAPS_CHAR_HEIGHT Specifies the default height (in pels) of the character box.

CAPS_CHAR_WIDTH Specifies the default width (in pels) of the character box.

CAPS_SMALL_CHAR_HEIGHT Specifies the default height (in pels) of the small character box. This number is zero if there is only one size of the character box.

CAPS_SMALL_CHAR_WIDTH Specifies the default width (in pels) of the small character box. This number is zero if there is only one size of the character box.

CAPS_COLORS Specifies the number of distinct colors supported at the same time, including reset (gray-scales count as distinct colors). If loadable color tables are supported, this is the number of entries in the device color table. For plotters, the value returned is the number of pens plus one (for the background).

CAPS_MOUSE_BUTTONS Specifies the number of mouse or tablet buttons that are available. A returned value of zero indicates that there are no mouse or tablet buttons available.

CAPS_FOREGROUND_MIX_SUPPORT Specifies the foreground-mix support. The possible values are as follows:

Value	Meaning
CAPS_FM_OR	OR
CAPS_FM_OVERPAINT	Overpaint
CAPS_FM_XOR	XOR
CAPS_FM_LEAVEALONE	Leave alone
CAPS_FM_AND	AND
CAPS_FM_GENERAL_BOOLEAN	Mixes 7 through 17

The value returned is the sum of the values appropriate to the mixes supported. A device capable of supporting the OR mix mode must, as a minimum, return 1 + 2 + 16 = 19, signifying support for the mandatory mix modes OR, overpaint, and "leave-alone." Note that these numbers correspond to the decimal representation of a bit string that is seven bits long, with each bit set to 1 if the appropriate mode is supported.

CAPS_BACKGROUND_MIX_SUPPORT Specifies the background mix support. The possible values are as follows:

Value	Meaning
CAPS_BM_OR	OR
CAPS_BM_OVERPAINT	Overpaint
CAPS_BM_XOR	XOR
CAPS_BM_LEAVEALONE	Leave alone

The value returned is the sum of the values appropriate to the mixes supported. A device must, as a minimum, return 2 + 16 = 18 signifying support for the mandatory background mixes overpaint and leave alone. Note that these numbers correspond to the decimal representation of a bit string that is five bits long, with each bit set to 1 if the appropriate mode is supported.

CAPS_LOADABLE_SYMBOL_SETS Specifies the number of fonts that may be loaded for **Vio**.

CAPS_WINDOW_BYTE_ALIGNMENT Specifies whether the client area of **Vio** windows should be byte-aligned. The possible values are as follows:

Value	Meaning
CAPS_BYTE_ALIGN_REQUIRED	Must be byte-aligned.
CAPS_BYTE_ALIGN_RECOMMENDED	More efficient if byte-aligned, but not required.
CAPS_BYTE_ALIGN_NOT_REQUIRED	Does not matter whether byte-aligned.

CAPS_BITMAP_FORMATS Specifies the number of bitmap formats supported by the device.

CAPS_RASTER_CAPS Specifies the raster-operations capability of the device. The possible values are as follows:

Value	Meaning
CAPS_RASTER_BITBLT	BitBlt supported
CAPS_RASTER_BANDING	Banding supported
CAPS_RASTER_BITBLT_SCALING	Scaling supported
CAPS_RASTER_SET_PEL	Set PEL support

CAPS_MARKER_WIDTH Specifies the default width (in pels) of the marker box.

CAPS_MARKER_HEIGHT Specifies the default depth (in pels) of the marker box.

CAPS_DEVICE_FONTS Specifies the number of device-specific fonts.

CAPS_GRAPHICS_SUBSET Specifies the graphics-drawing subset supported (3 indicates GOCA DR/3).

CAPS_GRAPHICS_VERSION Specifies the graphics-architecture version supported (1 indicates version 1).

CAPS_GRAPHICS_VECTOR_SUBSET Specifies the graphics-vector-drawing subset supported (2 indicates GOCA VS/2).

CAPS_GRAPHICS_CHAR_WIDTH Specifies the default **Gpi** character-box width (in pels).

CAPS_GRAPHICS_CHAR_HEIGHT Specifies the default **Gpi** character-box height (in pels).

CAPS_DEVICE_WINDOWING Specifies the support for device windows. This value may be CAPS_DEV_WINDOWING_SUPPORT if the device supports windowing.

CAPS_ADDITIONAL_GRAPHICS Specifies additional graphics support. The possible values are as follows:

Value	Meaning
CAPS_GRAPHICS_KERNING_SUPPORT	The device supports kerning.
CAPS_FONT_OUTLINE_DEFAULT	Outline font is the default.
CAPS_FONT_IMAGE_DEFAULT	Font image is the default.
CAPS_SCALED_DEFAULT_MARKERS	Scaled default markers.

CAPS_RESERVED Specifies the maximum number of distinct colors available at one time.

CAPS_PHYS_COLORS Specifies the maximum number of distinct colors that can be specified on the device.

CAPS_COLOR_INDEX Specifies the maximum logical-color-table index supported for the device. This value must be at least 7. For the EGA and VGA device drivers, the value is 63.

CAPS_COLOR_PLANES Specifies the number of color planes.

CAPS_COLOR_BITCOUNT Specifies the number of adjacent color bits for each pel (within one plane).

CAPS_COLOR_TABLE_SUPPORT Specifies the support for loadable color tables. It can be one of the following values:

Value	Meaning
CAPS_COLTABL_RGB_8	Set if the RGB color table can be loaded, with a minimum support of 8 bits each for red, green, and blue.
CAPS_COLTABLE_RGB_8_PLUS	Set if a color table with other than 8 bits for each primary color can be loaded.
CAPS_COLTABLE_TRUE_MIX	Set if true mixing occurs when the logical color table has been realized, providing that the size of the logical color table is not greater than the number of distinct colors supported (see CAPS_COLORS).
CAPS_COLTABL_REALIZE	Set if a loaded color table can be realized.

See Also

DevOpenDC

Changes

DevQueryCaps can also retrieve information about colors by using the following constants:

CAPS_COLOR_BITCOUNT
CAPS_COLOR_PLANES
CAPS_COLOR_TABLE_SUPPORT
CAPS_COLTABL_REALIZE
CAPS_COLTABL_RGB_8
CAPS_COLTABLE_RGB_8_PLUS
CAPS_COLTABLE_TRUE_MIX
CAPS_GRAPHICS_CHAR_WIDTH
CAPS_GRAPHICS_CHAR_HEIGHT

■ DosAllocHuge Change

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

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

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

The **DosAllocHuge** function can specify that segments can be shared by other processes. If the SEG_GETTABLE flag is used, other processes can gain access to the shared memory by calling the **DosGetSeg** function. If the SEG_GIVEABLE flag is used, the memory can be shared by other processes after the process allocating the memory has called the **DosGiveSeg** function. In both cases, the process allocating the memory must pass the selector to the process that will share the memory.

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

Parameters

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

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

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

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

fsAttr Specifies the segment attributes. This parameter can be one or more of the following values:

Value	Meaning
SEG_DISCARDABLE	Creates a discardable, nonsharable segment. Once the segment is unlocked, it may be discarded to satisfy another memory-allocation request.
SEG_GETTABLE	Creates a sharable segment that other processes can retrieve by using the **DosGetSeg** function.
SEG_GIVEABLE	Creates a sharable segment that the owning process can give to other processes by using the **DosGIveSeg** function.
SEG_NONSHARED	Creates a nonsharable, nondiscardable segment. This value cannot be combined with any other value.
SEG_SIZEABLE	Specifies that a shared segment can be reduced in size by **DosReallocSeg**.

Return Value

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

ERROR_NOT_ENOUGH_MEMORY

Comments

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

If necessary, the system will discard an unlocked discardable segment in order to satisfy another allocation request. The new allocation request can come from any process, including the process that allocated the segment being discarded.

The **DosFreeSeg** function frees all segments when passed the first selector. If the segments were declared as sharable, they will not be discarded from memory until the last process using them calls **DosFreeSeg**.

DosAllocHuge can be issued from ring 2, but the segments will be allocated as ring-3 segments.

Restrictions

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

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

Example

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

```
CHAR huge *pchBuffer;
SEL sel;
DosAllocHuge(2,                 /* number of segments             */
    200,                        /* size of last segment           */
    &sel,                       /* address of selector            */
    5,                          /* maximum segments for reallocation */
    SEG_NONSHARED);             /* sharing flag                   */
pchBuffer = MAKEP(sel, 0);      /* converts selector to pointer   */
```

See Also

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

Changes

SEG_SIZEABLE is a possible value for the *fsAttr* parameter. It allows a shared segment to be reduced in size by the **DosReallocHuge** function.

This request can be issued from ring 2, but the segment will be allocated as a ring-3 segment.

Corrections

The example incorrectly requested three 64K segments instead of the two described.

■ DosAllocSeg — Change

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

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

The **DosAllocSeg** function allocates a memory segment and copies the segment selector to a specified variable.

The **DosAllocSeg** function can specify that segments can be shared by other processes. If the SEG_GETTABLE flag is used, other processes can gain access to the shared memory by calling the **DosGetSeg** function. If the SEG_GIVEABLE flag is used, the memory can be shared by other processes after the process allocating the memory has called the **DosGiveSeg** function. In both cases, the process allocating the memory must pass the selector to the process that will share the memory.

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

Parameters

usSize Specifies the number of bytes to allocate. This number can be any value in the range 0 through 65,535. If this value is zero, the function allocates 65,536 bytes.

psel Points to the variable that receives the segment selector.

fsAttr Specifies the segment attributes. This parameter can be one or more of the following values:

Value	Meaning
SEG_DISCARDABLE	Creates a discardable, nonsharable segment. Once the segment is unlocked, it may be discarded to satisfy another memory-allocation request.
SEG_GETTABLE	Creates a sharable segment that other processes can retrieve by using the **DosGetSeg** function.
SEG_GIVEABLE	Creates a sharable segment that the owning process can give to other processes by using the **DosGiveSeg** function.
SEG_NONSHARED	Creates a nonsharable, nondiscardable segment. This value cannot be combined with any other value.
SEG_SIZEABLE	Specifies that a shared segment can be reduced in size by **DosReallocSeg**.

Return Value

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

ERROR_NOT_ENOUGH_MEMORY

Comments

If the SEG_DISCARDABLE attribute is set, the **DosAllocSeg** function automatically locks the segment. The segment cannot be discarded until the **DosUnLockSeg** function is called. Before a process accesses an unlocked discardable segment, it must call the **DosLockSeg** function to determine whether the segment has been discarded, and to prevent the segment from being discarded while it is accessing it.

If necessary, the system will discard an unlocked discardable segment in order to satisfy another allocation request. The new allocation request can come from any process, including the process that allocated the segment being discarded.

The **DosFreeSeg** function frees the segment. If the segment was declared as sharable, it will not be discarded from memory until the last process using it calls **DosFreeSeg**.

The **DosAllocSeg** function can allocate only up to 64K of contiguous memory. To allocate more than 64K, use the **DosAllocHuge** function.

DosAllocSeg can be issued from ring 2, but the segment will be allocated as a ring-3 segment.

Restrictions

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

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

Example

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

```
PCH pchBuffer;
SEL sel;

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

See Also

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

Changes

SEG_SIZEABLE is a possible value for the *fsAttr* parameter. It allows a shared segment to be reduced in size by the **DosReallocHuge** function.

This request can be issued from ring 2, but the segment will be allocated as a ring-3 segment.

DosAllocShrSeg — Change

```
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 specified variable.

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

Parameters

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

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

\sharemem\*name*

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

psel Points to the variable that receives the segment selector.

Return Value

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

ERROR_ALREADY_EXISTS
ERROR_INVALID_NAME
ERROR_NOT_ENOUGH_MEMORY

Comments

A process can allocate up to 256 shared segments. The number of segments that can be allocated may be less due to system usage at the time the allocation request is made.

The **DosFreeSeg** function frees the segment. The segment will not be discarded from memory until the last process using it calls **DosFreeSeg**.

DosAllocShrSeg can be issued from ring 2, but the shared-memory segment will be allocated as a ring-3 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 can use the memory if they know the name.

```
SEL sel;

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

See Also

DosAllocHuge, DosAllocSeg, DosFreeSeg, DosGetSeg, DosGetShrSeg, DosGiveSeg

Changes

The number of segments a process can allocate has been increased to approximately 256 (the actual number varies according to system usage).

The error message ERROR_INVALID_HANDLE has been changed to ERROR_INVALID_NAME.

■ DosCopy — New

```
USHORT DosCopy(pszSrc, pszDest, usOpt, ulReserved)
PSZ pszSrc;             /* pointer to name of source file  */
PSZ pszDest;            /* pointer to name of target file  */
USHORT usOpt;           /* options                         */
ULONG ulReserved;       /* must be zero                    */
```

The **DosCopy** function copies a file or subdirectory.

Parameters

pszSrc Points to the null-terminated string that specifies the file or directory to copy. This string must be a valid MS OS/2 filename and cannot contain wildcard characters.

pszDest Points to the null-terminated string that specifies the name of the file, directory, or device to copy the value of *pszSrc* to. This string must be a valid MS OS/2 filename and cannot contain wildcard characters.

usOpt Specifies an option that can be used in the copy operation (it is ignored if the destination is a device). This parameter can be one of the following values:

Value	Meaning
DCPY_EXISTING	Copy the source file to the destination file, even if the destination file already exists. If neither this option nor the DCPY_APPEND option is specified, and the file exists, the value ERROR_ACCESS_DENIED is returned.
DCPY_APPEND	Append the data in the source file to the end of the destination file. If the destination file does not exist, a new file is created.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_DIRECTORY
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_FILENAME_EXCED_RANGE
ERROR_INSUFFICIENT_DISK_SPACE
ERROR_INVALID_PARAMETER
ERROR_NOT_DOS_DISK
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION

Comments

The **DosCopy** function can be used to copy individual files or entire directories (including any subdirectories within the directory). The source and destination files can be on different drives.

If an I/O error occurs when a file is being copied, the destination file is deleted from the destination directory unless the DCPY_APPEND option is specified. In this case, the destination file is restored to its original size.

The **DosCopy** function copies the attributes of the source to the destination file, except when appending to an existing file.

You cannot specify only the drive as the destination. You must give the path on the drive where the file or directory is to be copied.

Example

This example copies the directory *xyz* from drive C, including its files and subdirectories, to the root directory on drive A.

```
DosCopy("c:\\xyz",         /* source directory        */
    "a:\\",                /* destination directory   */
    DCPY_EXISTING,         /* replaces existing files */
    OL);                   /* reserved                */
```

See Also

DosMove

■ DosCreateSem — Correction

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

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

Parameters

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

phssm Points to the variable that receives the semaphore handle.

pszSemName Points to a null-terminated string that identifies the semaphore. The string must have the form **\sem\***name*. The string name, *name*, must have the same format as an MS OS/2 filename and must be unique.

Return Value

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

ERROR_ALREADY_EXISTS
ERROR_INVALID_NAME
ERROR_INVALID_PARAMETER
ERROR_TOO_MANY_SEMAPHORES

Comments

The process calling **DosSemCreate** receives exclusive ownership of the semaphore if the CSEM_PRIVATE flag is set in the *fExclusive* parameter. Exclusive ownership prevents other processes from setting or clearing the semaphore. Other processes can open the semaphore and wait for it to change status, but they cannot change its status. Another process can obtain ownership of the semaphore, however, by calling the **DosSemRequest** function. If ownership of the semaphore changed through **DosSemRequest**, the process that originally called **DosCreateSem** no longer has ownership. It cannot change the status of the semaphore until it regains ownership by calling **DosSemRequest.**

Example

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

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

See Also

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

Corrections

The comments incorrectly indicated that the semaphore is always owned by the process that calls **DosCreateSem**. The semaphore is owned by the calling process only if the CSEM_PRIVATE flag is set in the *fExclusive* parameter.

■ DosCreateThread Correction

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

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

Parameters

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

ptidThread Points to the variable that receives the thread identifier.

pbThrdStack Points to the stack of the new thread.

Return Value

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

ERROR_NO_PROC_SLOTS
ERROR_NOT_ENOUGH_MEMORY

Comments

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

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

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

Because 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 can use the corresponding handles to access the given item.

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

Before calling the **DosCreateThread** function, set the **es** register to zero or assign to it a selector that will remain valid for the duration of the new thread. If you fail to set the **es** register to one of these values, the thread may unexpectedly terminate as a result of a general protection fault.

Example

This example sets aside a 2K buffer to be used as stack space for any threads created. The first stack is set at the end of the array. The thread is created by calling the **DosCreateThread** function. The thread terminates by calling the **DosExit** function.

```
VOID FAR Thread1();
BYTE abStackArea[2048];
    .
    .
    .
    PVOID pStack1 = abStackArea + sizeof(abStackArea);
    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 **DosAllocSeg, DosExit, DosResumeThread, DosSuspendThread**

Corrections The example indicated that a 512K-byte stack was allocated. This has been changed to a 2K-byte stack.

■ DosDevIOCtl2 New

USHORT DosDevIOCtl2(*pvData*, *cbData*, *pvParmList*, *cbParmList*, *usFunct*, *usCat*, *hDev* **)**

```
PVOID pvData;          /* pointer to buffer for data     */
USHORT cbData;         /* length of data buffer          */
PVOID pvParmList;      /* pointer to list of parameters  */
USHORT cbParmList;     /* length of parameter list       */
USHORT usFunct;        /* function code                  */
USHORT usCat;          /* device category                */
HFILE hDev;            /* device handle                  */
```

The **DosDevIOCtl2** function performs control functions on the device specified by the file or device handle.

Parameters

pvData Points to a data buffer.

cbData Specifies the length (in bytes) of the data buffer.

pvParmList Points to an argument list for a specified command.

cbParmList Specifies the length (in bytes) of the argument list for a specified command.

usFunct Specifies a function code for a specified device. This parameter can be any value from 0 through 255.

usCat Specifies a device category. This parameter can be any value from 0 through 255.

hDev Identifies the device. This handle must have been created previously by using the **DosOpen** function.

Return Value

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

ERROR_INVALID_CATEGORY
ERROR_INVALID_DRIVE
ERROR_INVALID_FUNCTION
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER

Comments

This function provides a way for a program to implement a customized **IOCtl** function.

If the *pvData* parameter is zero, this parameter is not defined for the **IOCtl** function being specified, and the value passed in the *cbData* parameter is ignored.

If the *pvParmList* parameter is zero, this parameter is not defined for the **IOCtl** function being specified, and the value passed in the *cbParmList* parameter is ignored.

Whenever the *pvData* or *pvParmList* parameter is a value other than zero, the associated length parameter cannot be zero. The length parameters are not passed to device drivers that do not support them.

See Also

DosDevIOCtl

■ DosEditName New

```
USHORT DosEditName( usEditLevel, pszSrc, pszEdit, pszDst, cbDst)
USHORT usEditLevel;     /* edit level               */
PSZ pszSrc;             /* pointer to source string */
PSZ pszEdit;            /* pointer to editing string */
PBYTE pszDst;           /* pointer to target buffer */
USHORT cbDst;           /* length of target buffer  */
```

The **DosEditName** function copies a source string to a revised destination string by using an editing string and rules for converting wildcard characters.

Parameters

usEditLevel Specifies the version of editing semantics to use in changing the copy of the source string. (Editing semantics are the rules used by the system to convert wildcard characters.) For MS OS/2, version 1.2, this parameter must be 0x0001.

pszSrc Points to the null-terminated string to copy. The string should contain only the component of the path to be edited, not the entire path.

pszEdit Points to the null-terminated string to use for editing.

pszDst Points to the buffer that contains the new string.

cbDst Specifies the length (in bytes) of the buffer pointed to by the *pszDst* parameter.

Return Value

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

ERROR_INVALID_NAME
ERROR_INVALID_PARAMETER

Comments

For MS OS/2, version 1.2, the destination string is always converted to uppercase.

The **DosEditName** function is typically used in copy and rename/move operations.

Example

This example takes the source name *abc.txt* and an editing string of **.doc* and calls **DosEditName** to produce the string *ABC.DOC*:

```
CHAR szDst[14];

DosEditName(1, "abc.txt", "*.doc", szDst, sizeof (szDst));
```

■ DosEnterCritSec — Change

USHORT DosEnterCritSec(*VOID* **)**

The **DosEnterCritSec** function suspends execution of all threads in the current process, except for the calling thread. Suspended threads cannot execute until the current thread calls the **DosExitCritSec** function.

This function has no parameters.

Return Value

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

ERROR_CRITSEC_OVERFLOW

Comments

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

MS OS/2 maintains the number of outstanding **DosEnterCritSec** requests. This count is incremented by **DosEnterCritSec** requests and decremented by **DosExitCritSec** requests. If the count is greater than zero, a **DosExitCritSec** request will not restore normal thread execution. If the count exceeds 65535, the error ERROR_CRITSEC_OVERFLOW will be returned.

See Also

DosCreateThread, DosExitCritSec, DosHoldSignal, DosSetSigHandler

Changes

DosEnterCritSec now returns zero if the function is successful. Otherwise, it returns an error value. It did not return a value in earlier versions.

For MS OS/2, version 1.2, a count is maintained of the number of times **DosEnterCritSec** is called. Normal thread execution is not restored until an equal number of calls are made to **DosExitCritSec**.

■ DosEnumAttribute New

```
USHORT DosEnumAttribute( usRefType, pvFile, ulEntry, pvBuf, cbBuf, pulCount, ulInfoLevel, ulReserved)
USHORT usRefType;     /* reference type              */
PVOID pvFile;         /* filename/handle             */
ULONG ulEntry;        /* starting entry in list      */
PVOID pvBuf;          /* data buffer                 */
ULONG cbBuf;          /* buffer size                 */
PULONG pulCount;      /* number of entries to return */
ULONG ulInfoLevel;    /* info level                  */
ULONG ulReserved;     /* reserved                    */
```

The **DosEnumAttribute** function enumerates extended attributes for a specified file or subdirectory.

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

Parameters

usRefType Specifies whether the *pvFile* parameter points to a file handle or to a string that contains a file or directory name. This parameter can be one of the following values:

Value	Meaning
ENUMEA_REFTYPE_FHANDLE	A handle
ENUMEA_REFTYPE_PATH	File or directory name

pvFile Points to the handle obtained from the **DosOpen** or **DosOpen2** function or to a null-terminated string that contains a file or directory name.

ulEntry Specifies where to start enumerating extended attributes. A value of 1 specifies the first attribute for the file.

pvBuf Points to the buffer that receives the extended attributes. For a ENUMEA_LEVEL_NO_VALUE-level request, the buffer is in the form of a **DENA1** structure that contains only the names of the extended attributes. The **DENA1** structure has the following form:

```
typedef struct _DENA1 {
    UCHAR  reserved;
    UCHAR  cbName;
    USHORT cbValue;
    UCHAR  szName[1];
} DENA1;
```

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

cbBuf Specifies the length (in bytes) of the buffer pointed to by the *pvBuf* parameter.

pulCount Points to the variable that specifies the number of extended attributes requested and, on return, contains the number retrieved. A value of 0xFFFFFFFF returns as many extended attributes as will fit in the supplied buffer.

ulInfoLevel Specifies the information level requested. For MS OS/2, version 1.2, the only possible value is ENUMEA_LEVEL_NO_VALUE.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_FILENAME_EXCED_RANGE
ERROR_INVALID_HANDLE
ERROR_ACCESS_DENIED
ERROR_PATH_NOT_FOUND
ERROR_NOT_ENOUGH_MEMORY
ERROR_INVALID_LEVEL
ERROR_INVALID_PARAMETER
ERROR_BUFFER_OVERFLOW

Comments

The order in which attributes are returned may not be the same if the **Dos-EnumAttribute** function is called a second time because other threads or processes may have changed the order.

Example

This example allocates 1K of memory for the extended-attribute names, calls **DosEnumAttribute** to retrieve the extended-attribute names for the file *eafile*, and then displays the names one at a time:

```
#define BUFSIZE

SEL sel;
PDENA1 pdenal;
ULONG ulCount;
USHORT offset = 0;

DosAllocSeg(BUFSIZE, &sel, SEG_NONSHARED); /* allocates buffer      */
pdenal = MAKEP(sel, 0);          /* initializes pointer to buffer    */
ulCount = 0xFFFFFFFF;
if (!DosEnumAttribute(ENUMEA_REFTYPE_PATH,          /* path supplied */
        "eafile",                                   /* filename      */
        1L,                 /* starts enum. with first attr.         */
        pdenal,             /* buffer address                        */
        BUFSIZE,            /* buffer size                           */
        &ulCount,           /* number of attributes to retrieve      */
        ENUMEA_LEVEL_NO_VALUE, /* type of request                    */
        0L)) {              /* reserved                              */
    while (ulCount--) {     /* while there are attribute names       */
        VioWrtTTY(pdenal->szName, (USHORT) pdenal->cbName, 0L);
        VioWrtTTY("\r\n", 2, 0L);
        offset += sizeof(DENA1) + pdenal->cbName;
        pdenal = MAKEP(sel, offset);          /* points to next name */
    }
}
```

■ DosExitCritSec Change

USHORT DosExitCritSec(*VOID* **)**

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

This function has no parameters.

Return Value

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

ERROR_CRITSEC_UNDERFLOW

Comments

MS OS/2 maintains the number of outstanding **DosEnterCritSec** requests. This count is incremented by **DosEnterCritSec** requests and decremented by **DosExitCritSec** requests. If the count is greater than zero, a **DosExitCritSec** request will not restore normal thread execution. If the count is less than zero, the ERROR_CRITSEC_UNDERFLOW will be returned.

See Also

DosCreateThread, DosEnterCritSec

Changes

DosExitCritSec now returns an error value if it is called without a corresponding call to **DosEnterCritSec.**

■ DosExitList — Correction

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

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

Parameters

fFnCode Specifies whether a function's address is added to or removed from the list. If the function is added, the high byte of this parameter specifies the order in which the function should be called. The exit-list routines with a low-order high byte will be called before those with a high-order high byte. The low byte of this parameter can be one of the following values:

Value	Meaning
EXLST_ADD	Adds the function to the termination list. If this flag is specified, the high byte of the parameter specifies the order in which the function is called. It can be a value from 0 through 255. A value of 0 specifies that this function is to be called first. In the event of duplicate order numbers, the last function added with the duplicate order number is called before the first function added with the duplicate order number.
EXLST_EXIT	Termination processing is complete. Calls the next function on the termination list.
EXLST_REMOVE	Removes the function from the termination list.

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

Return Value

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

ERROR_INVALID_DATA
ERROR_NOT_ENOUGH_MEMORY

Comments

When adding an exit-list function, it is important that the exit-list function not call any system functions with a lower exit-list order. The order is determined by the high-byte of the *fFnCode* parameter. The following list defines the orders of the various system components:

Order	Component
0x80–0x88	Extended Edition Database Manager
0x90–0x98	Extended Edition Communication Manager
0xA0–0xA8	Presentation Manager
0xB0	KBD component
0xC0	VIO component
0xD0	IPC Queues component

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

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

```
VOID PASCAL FAR FuncName(usTermCode)
USHORT usTermCode;
{
    .
    .
    .
    DosExitList(EXLST_EXIT, NULL);
}
```

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

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

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

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

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

Example

This example calls **DosExitList**, which then adds the locally defined function CleanUp to the list of routines to be called when the process terminates.

The CleanUp function displays a message that it is cleaning up, and then calls **DosExitList**, reporting that it has finished and that the next function on the termination list can be called.

```
    /* Add the function, and have it be called last. */

    DosExitList(EXLST_ADD | 0xFF00, CleanUp);
    .
    .
    .
    DosExit(EXIT_PROCESS, 0);
}

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

See Also

DosCreateThread, DosExecPgm, DosExit, DosKillProcess

Corrections

When the EXLST_ADD constant is used in the *fFnCode* parameter, the high byte of the parameter contains an order number (0 through 255). You can use this number to specify the order in which your exit-list function is called.

The function template in the example incorrectly listed the prototype of the termination function as PFNEXITLIST. It should be VOID PASCAL FAR.

■ DosFileIO — New

USHORT DosFileIO(*hf*, *pbCmd*, *cbCmd*, *pusErr* **)**

HFILE *hf*;	/* file handle */
PBYTE *pbCmd*;	/* pointer to buffer for commands */
USHORT *cbCmd*;	/* length of command buffer */
PUSHORT *pusErr*;	/* pointer to error offset */

The **DosFileIO** function performs multiple lock, unlock, seek, read, and write operations on a file.

Parameters

hf Identifies the file on which to perform the commands. This handle must have been created previously by using the **DosOpen** function.

pbCmd Points to the buffer that contains one or more of the following structures: **FIOLOCKCMD, FIOLOCKREC, FIOUNLOCKCMD, FIOUNLOCKREC, FIOSEEKCMD,** or **FIOREADWRITE.** The structures have the following forms:

```
typedef struct _FIOLOCKCMD {
    USHORT usCmd;
    USHORT cLockCnt;
    ULONG  cTimeOut;
} FIOLOCKCMD;

typedef struct _FIOLOCKREC {
    USHORT fShare;
    ULONG  cbStart;
    ULONG  cbLength;
} FIOLOCKREC;
```

```
typedef struct  _FIOUNLOCKCMD {
    USHORT usCmd;
    USHORT cUnlockCnt;
} FIOUNLOCKCMD;

typedef struct  _FIOUNLOCKREC {
    ULONG cbStart;
    ULONG cbLength;
} FIOUNLOCKREC;

typedef struct  _FIOSEEKCMD {
    USHORT usCmd;
    USHORT fsMethod;
    ULONG  cbDistance;
    ULONG  cbNewPosition;
} FIOSEEKCMD;

typedef struct  _FIOREADWRITE {
    USHORT usCmd;
    PVOID  pbBuffer;
    USHORT cbBufferLen;
    USHORT cbActualLen;
} FIOREADWRITE;
```

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

cbCmd Specifies the length (in bytes) of the *pbCmd* parameter.

pusErr Points to a variable that receives the byte offset of the structure that caused an error. The offset is relative to the beginning of the buffer pointed to by the *pbCmd* parameter.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_DIRECT_ACCESS_HANDLE
ERROR_INTERRUPT
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_LOCK_VIOLATION
ERROR_NEGATIVE_SEEK
ERROR_SEEK_ON_DEVICE
ERROR_SHARING_BUFFER_EXCEEDED

Comments

The **DosFileIO** function allows you to combine the following operations into a single function call:

- Locking and unlocking multiple file ranges
- Changing the file-position pointer
- Reading and/or writing

Combining these operations into one call can improve system performance, particularly in a networking environment.

The **DosFileIO** function provides a simple mechanism for denying other processes read/write or write access to regions of the file. If another process attempts to read from or write to a no-access region, or attempts to write in a read-only region, an error is returned. If a time-out occurs before the locking operation is complete, **DosFileIO** returns an error to the calling process.

Since the calling process may return after the time-out period has expired without receiving an ERROR_SEM_TIMEOUT message, semaphore time-out values should not be used for exact timing or for determining the sequence of I/O operations.

Before a range is locked, it must be cleared of any locked subranges or locked overlapping ranges.

Each I/O operation completes before the next one begins. The operations continue until all are complete or until one fails.

Example

This example opens the file *abc.txt*, allocates memory for the command buffer, initializes the commands in that buffer, and calls **DosFileIO** to move the file 10 bytes into the file and then read from the file:

```
HFILE hf;
USHORT usAction;
SEL sel;
BYTE abBuf[512];
LONG lError;

PFIOREADWRITE pfiorw;
PFIOSEEKCMD pfioseek;

DosOpen("abc.txt", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
DosAllocSeg(sizeof(FIOSEEKCMD) + sizeof(FIOREADWRITE),
    &sel, SEG_NONSHARED);

pfioseek = MAKEP(sel, 0);
pfioseek->usCmd = FIO_SEEK;
pfioseek->fsMethod = FILE_BEGIN;
pfioseek->cbDistance = 10L;

pfiorw = MAKEP(sel, sizeof(FIOSEEKCMD));
pfiorw->usCmd = FIO_READ;
pfiorw->pbBuffer = (PVOID) abBuf;
pfiorw->cbBufferLen = sizeof(abBuf);

DosFileIO(hf,                                        /* file handle    */
    MAKEP(sel, 0),                                   /* buffer address */
    (sizeof(FIOSEEKCMD) + sizeof(FIOREADWRITE)),     /* buffer size    */
    &lError);                             /* address of error variable */
```

See Also

DosChgFilePtr, DosFileLocks, DosOpen, DosRead, DosWrite

DosFindFirst2 New

USHORT DosFindFirst2(*pszFileName*, *phDir*, *usAttribute*, *pBuf*, *cbBuf*, *pusSearchCount*, *usInfoLevel*, *ulReserved* **)**

```
PSZ pszFileName;           /* pointer to filename              */
PHDIR phDir;               /* pointer to directory handle      */
USHORT usAttribute;        /* attributes of file to be found   */
PVOID pBuf;                /* pointer to buffer for results    */
USHORT cbBuf;              /* size of results buffer           */
PUSHORT pusSearchCount;    /* number of entries found          */
USHORT usInfoLevel;        /* level of information to retrieve */
ULONG ulReserved;          /* must be zero                     */
```

The **DosFindFirst2** function searches a directory for the file or files whose filename and attributes match the specified filename and attributes.

The **DosFindFirst2** function is a family API function.

Parameters

pszFileName Points to a null-terminated string. This string must be a valid MS OS/2 path and can contain wildcard characters.

phDir Points to the variable that contains the handle of the directory to search.

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

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

pBuf Points to the buffer in which the file information is returned. The format for this buffer is determined by the value specified in the *usInfoLevel* parameter.

cbBuf Specifies the size (in bytes) of the buffer pointed to by *pBuf*.

pusSearchCount Points to the variable that specifies the number of matching entries to locate. The **DosFindFirst2** function copies the number of entries found to this parameter before returning.

usInfoLevel Specifies the type of file information to retrieve. This parameter can be one of the following values:

Value	Meaning
FIL_STANDARD	Return a **FILEFINDBUF** structure with the results of the search. The information returned is identical to that returned by the **DosFindFirst** function.
FIL_QUERYEASIZE	Return a **FILEFINDBUF2** structure with the results of the search, and that contains the size of the buffer needed to retrieve the extended attributes.
FIL_QUERYEASFROMLIST	Return a buffer that contains both the file information and the extended attributes for the file.

The **FILEFINDBUF** structure has the following form:

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

The **FILEFINDBUF2** structure has the following form:

```
typedef struct _FILEFINDBUF2 {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
    USHORT cbList;
    UCHAR  cchName;
    CHAR   achName[13];
} FILEFINDBUF2;
```

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

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_BUFFER_OVERFLOW
ERROR_EAS_DIDNT_FIT
ERROR_EA_LIST_INCONSISTENT
ERROR_FILENAME_EXCED_RANGE
ERROR_INVALID_EA_NAME
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_META_EXPANSION_TOO_LONG
ERROR_NO_MORE_FILES
ERROR_NO_MORE_SEARCH_HANDLES
ERROR_PATH_NOT_FOUND

Comments

The **DosFindNext** function uses the directory handle pointed to by the *phDir* parameter of the **DosFindFirst2** function to repeat the search. If **DosFindFirst2** returns an error value other than ERROR_EAS_DIDNT_FIT, no directory handle is allocated.

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

If the value of the *usInfoLevel* parameter is FILE_QUERYEASIZE, the **cbList** field of the **FILEFINDBUF2** structure can be used to calculate the size of the buffer necessary for a FILE_QUERYEASFROMLIST information request. For MS OS/2 version 1.2, the value of **cbList** will never exceed 65,535.

To use a FILE_QUERYEASFROMLIST information request, you must supply a buffer large enough for an **EAOP** structure and a **FILEFINDBUF** structure, plus enough space for the the extended attributes. You must initialize the first portion of this buffer as an **EAOP** structure, and fill in the **GEALIST** structure with the extended-attribute names to retrieve. On return, the **EAOP** structure will be unchanged. It will be followed immediately by a **FILEFINDBUF2** structure, without the last three fields. This is followed by an **FEALIST** structure (the address is the same as the **cbList** field of the **FILEFINDBUF2** structure). The **FEALIST** structure is in turn followed by a single byte that specifies the length of the filename, and that is followed by a null-terminated string that specifies the filename. For an example of how to use structure pointers to access each of these fields, see the "Example" section.

If there is not enough room in the output buffer to hold the extended-attribute information, the error ERROR_EAS_DIDNT_FIT is returned. The search handle will be allocated, however, and can be used in subsequent calls to the **DosFindNext** function. If no extended attribute is found, the **FEA** structure for that extended attribute will contain the name of the attribute, but the **cbValue** field will be zero.

Example

This example shows how to set up pointers to access the various fields of the buffer returned by a FIL_QUERYEASFROMLIST level request:

```
/* Declare a structure to retrieve the .TYPE attribute name. */

typedef struct _TYPEATTR {
    ULONG cbList;
    BYTE  cbName;
    CHAR  szName[6];
} TYPEATTR;

#define BUFSIZE 2 * 1024              /* default buffer size          */

SEL sel;                              /* selector for buffer          */
HDIR hdir = HDIR_CREATE;              /* directory handle             */
USHORT usSearchCount = 1;             /* number of files to retrieve  */
TYPEATTR typeattr;                    /* TYPE attribute structure     */
PEAOP peaop;
PFILEFINDBUF2 pfindbuf2;
PFEALIST pfeal;
PSZ pszFileName;
PUCHAR pcchFileName;

DosAllocSeg(BUFSIZE, &sel, SEG_NONSHARED); /* creates buffer          */
peaop = MAKEP(sel, 0);                     /* sets up peaop pointer   */

typeattr.cbList = sizeof(TYPEATTR);        /* structure size          */
strcpy(typeattr.szName, ".TYPE");          /* EA name                 */
typeattr.cbName = sizeof(typeattr.szName) - 1; /* name length         */

peaop->fpGEAList = (PGEALIST) &typeattr;   /* size of GEALIST struc.  */

if (!DosFindFirst2("eafile", &hdir, FILE_NORMAL,
        peaop, BUFSIZE,
        &usSearchCount, FIL_QUERYEASFROMLIST, 0L)) {
    pfindbuf2 = MAKEP(sel, sizeof(EAOP));   /* FILEFINDBUF structure */
    pfeal = (PFEALIST) &pfindbuf2->cbList;  /* FEALIST structure     */
    pcchFileName = ((PSZ) pfeal) + pfeal->cbList; /* filename length */
    pszFileName = pcchFileName + 1;               /* filename        */
}
```

See Also

DosFindClose, DosFindFirst, DosFindNext, DosQFileMode, DosQFSInfo

■ DosFindNext Change

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

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

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

Parameters

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

pfindbuf Points to the structure that receives the result of the search. This structure will be either a **FILEFINDBUF** or **FILEFINDBUF2** structure, depending on the information level requested in the **DosFindFirst** or **DosFindFirst2** function that preceded this function. For specific information on the format of these structures, see the **DosFindFirst** and **DosFindFirst2** functions.

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

pcSearch Points to the variable that specifies the number of matching filenames to locate. The function copies the number of filenames found to the variable before returning.

Return Value

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

ERROR_BUFFER_OVERFLOW
ERROR_INVALID_HANDLE
ERROR_INVALID_PARAMETER
ERROR_NO_MORE_FILES
ERROR_NOT_DOS_DISK
ERROR_EAS_DIDNT_FIT

Comments

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

Restrictions

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

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

Example

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

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

See Also **DosBufReset, DosClose, DosFindClose, DosFindFirst, DosFindFirst2**

Changes **DosFindNext** returns the same type of structure as requested by the most recent call to either **DosFindFirst** or **DosFindFirst2.**

■ DosFreeResource **New**

```
USHORT DosFreeResource(pvData)
PVOID pvData;     /* pointer to data to free */
```

The **DosFreeResource** function frees memory allocated by a previous call to the **DosGetResource2** function.

Parameters *pvData* Points to the buffer to free. This pointer should have been returned by a previous call to the **DosGetResource2** function.

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

See Also **DosGetResource, DosGetResource2**

■ DosFreeSeg **Change**

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

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

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

Parameters *sel* Specifies the segment to free.

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

ERROR_ACCESS_DENIED

Comments **DosFreeSeg** can be issued from ring 2, but the segment to free must be a ring-3 segment.

DosFreeSeg should not be used to free resource segments allocated by the **DosGetResource2** function. To free those segments, use the **DosFreeResource** function.

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

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

Example

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

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

See Also

DosAllocHuge, DosAllocSeg, DosAllocShrSeg, DosCreateCSAlias, DosFreeResource, DosGetResource, DosGetResource2

Changes

DosFreeSeg should not be used to free segments allocated by the **DosGetResource2** function.

■ DosFSAttach New

```
USHORT DosFSAttach(pszDevName, pszFSD, pData, cbData, fsOp, ulReserved)
PSZ pszDevName;       /* pointer to device name                       */
PSZ pszFSD;           /* pointer to file system                       */
PBYTE pData;          /* pointer to buffer for file-system arguments  */
USHORT cbData;        /* length of argument buffer                    */
USHORT fsOp;          /* attach or detach connection                  */
ULONG ulReserved;     /* must be zero                                 */
```

The **DosFSAttach** function attaches or detaches a drive or pseudo-character device from a remote file system.

Parameters

pszDevName Points to a null-terminated string that specifies the drive letter followed by a colon or a pseudo-character device name. If this parameter is a pseudo-character device name, the format of the string is \DEV\filename, where *filename* is a valid MS OS/2 filename.

pszFSD Points to a null-terminated string that specifies the name of the remote file system to attach to or detach from the device specified by the *pszDevName* parameter.

pData Points to a buffer that contains the file-system arguments. The meaning of the arguments is specific to the file system. The first word of the buffer specifies the number of strings it contains; the rest of the buffer contains contiguous strings.

cbData Specifies the length (in bytes) of the data buffer.

fsOp Specifies the type of operation to perform. A value of FS_ATTACH attaches a file-system connection. A value of FS_DETACH detaches a file-system connection.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_ALREADY_ASSIGNED
ERROR_INVALID_DRIVE
ERROR_INVALID_FSD_NAME
ERROR_INVALID_LEVEL
ERROR_INVALID_PATH
ERROR_NOT_ENOUGH_MEMORY

Comments

Drive letters that represent local drives cannot be redirected.

When a drive is attached to a file system, all requests to that drive are routed to the file system. When a drive is detached from a file system, the drive name can no longer be used.

When a pseudo-character device name is attached to a file system, all requests to that name are routed to the file system. When a pseudo-character device is detached from a file system, the device name can no longer be used unless it overlaid the name of an existing device; in this case, the previous device regains control.

Example

This example calls **DosFSAttach** to attach a LAN server to drive X, and then calls **DosFSAttach** again to detach the LAN server:

```
CHAR szShareName[] = { 1, 0,      /* number of strings           */
    "\\SERVER\SHARE" };           /* name of server and share point */

DosFSAttach("X:", "LAN", szShareName, sizeof(szShareName),
        FS_ATTACH, 0L);
    .
    .
    .
DosFSAttach("X:", "LAN", szShareName, sizeof(szShareName),
        FS_DETACH, 0L);
```

See Also

DosFSCtl

DosFSCtl — New

USHORT DosFSCtl(*pbData, cbData, pcbData, pbParms, cbParm, pcbParm, usFunct, pszRoute, hf, usRouteMethod, ulReserved***)**

```
PBYTE pbData;             /* pointer to data buffer              */
USHORT cbData;            /* buffer length                       */
PUSHORT pcbData;          /* pointer to buffer for actual length */
PBYTE pbParms;            /* pointer to parameter list           */
USHORT cbParm;            /* size of parameter list              */
PUSHORT pcbParm;          /* pointer to buffer for actual length */
USHORT usFunct;           /* function code                       */
PSZ pszRoute;             /* pointer to file-system name         */
HFILE hf;                 /* file or device handle               */
USHORT usRouteMethod;     /* routing method                      */
ULONG ulReserved;         /* must be zero                        */
```

The **DosFSCtl** function is used to call functions provided in a file system that are not part of the standard set of I/O functions.

Parameters

pbData Points to the buffer that receives data from the nonstandard function.

cbData Specifies the length (in bytes) of the buffer pointed to by the *pbData* parameter. If this value is not at least as large as the value pointed to by the *pcbData* parameter, the system returns the ERROR_BUFFER_OVERFLOW error value and the value pointed to by *pcbData* will contain the correct length.

pcbData Points to the variable that receives the actual length of data returned.

pbParms Points to a list of command-specific parameters.

cbParm Specifies the length (in bytes) of the *pbParms* parameter. If the buffer size is insufficient, the error value ERROR_BUFFER_OVERFLOW will is returned and *pcbParm* will contain the size of buffer needed.

pcbParm Points to the variable that contains the length of the commands passed to the function and, on return, contains the length of the commands returned by the function. *usFunct* Specifies a function code specific to the file system. This parameter can be one of the following values:

Value	Meaning
0x0000–0x7FFF	Reserved for MS OS/2.
0x8000–0xBFFF	Functions to be handled by local file systems.
0xC000–0xFFFF	Functions to be handled by remote file systems.

pszRoute Points to the string that contains the name of the file system or the path of a file or directory that the operation applies to.

hf Identifies the file or device.

usRouteMethod Specifies how the request will be routed. This parameter can be one of the following values:

Value	Meaning
FSCTL_HANDLE	Route via the file handle. The *pszRoute* parameter must be NULL, and the *hf* parameter must be a valid file or device handle.
FSCTL_PATHNAME	Route via a path. The *hf* parameter must be - 1, and the *pszRoute* parameter must be a valid MS OS/2 path.
FSCTL_FSDNAME	Route via a file-system name. The *hf* parameter must be - 1 and the *pszRoute* parameter must point to the name of a valid file system.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_BUFFER_OVERFLOW
ERROR_INTERRUPT
ERROR_INVALID_CATEGORY
ERROR_INVALID_FSD_NAME
ERROR_INVALID_FUNCTION
ERROR_INVALID_HANDLE
ERROR_INVALID_LEVEL
ERROR_INVALID_PARAMETER
ERROR_NOT_SUPPORTED

Comments

A *usFunct* value of 0x0001 returns new error code information from the file system; a value of 0x0002 returns the maximum size of individual extended attributes in the first word of the buffer pointed to by *pbData* and the maximum size of the full extended-attribute list in the second word of the buffer.

See Also

DosFSAttach

■ DosGetDBCSEv — Correction

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

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

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

Parameters

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

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

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

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

Return Value

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

ERROR_NLS_BAD_TYPE
ERROR_NLS_NO_COUNTRY_FILE
ERROR_NLS_NO_CTRY_CODE
ERROR_NLS_OPEN_FAILED
ERROR_NLS_TABLE_TRUNCATED
ERROR_NLS_TYPE_NOT_FOUND

Comments

The DBCS environment vector defines the low and high ranges for the DBCS lead-byte values.

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

```
BYTE low1, high1;
BYTE low, high2;
     .
     .
     .
BYTE lown, highn;
BYTE 0, 0;
```

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

Restrictions

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

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

See Also

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

Corrections

The **DosGetDBCSEv** function returns only the range for the lead byte of the character set, not for the range of the trail byte.

DosGetModHandle — Correction

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

The **DosGetModHandle** function retrieves the handle of a dynamic-link module. The **DosGetModHandle** function is typically used to make sure that a module has been loaded into memory. If the module has not been loaded, the function returns ERROR_MOD_NOT_FOUND, and the **DosLoadModule** function must be used to load the module.

Parameters

pszModName Points to the null-terminated string that specifies the module name. This string must be a valid MS OS/2 filename. If it does not specify a path and the filename extension, the function appends the default extension (*.dll*) and searches for the dynamic-link module in the directories specified by the **libpath** command in the *config.sys* file.

phMod Points to the variable that receives the module handle.

Return Value

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

ERROR_INTERRUPT
ERROR_MOD_NOT_FOUND

Example

This example calls **DosGetModHandle** to determine if the dynamic-link module *mydll.dll* is currently in memory. If *mydll.dll* is not in memory, **DosGetModHandle** calls **DosLoadModule** to load it. It then calls **DosGetModName** to get the full path of the module. (This example is accurate if *mydll.dll* exists in a directory defined by the **libpath** parameter of the *config.sys* file.)

```
USHORT usError;
HMODULE hmod;
CHAR achFailName[128], szModName[128];

if (usError = DosGetModHandle("mydll", &hmod)) {
    if (usError == ERROR_MOD_NOT_FOUND)
        DosLoadModule(achFailName, sizeof(achFailName),
            "mydll", &hmod);
}
DosGetModName(hmod, sizeof(szModName), szModName);
```

See Also

DosFreeModule, DosGetModName, DosLoadModule

Corrections

If the *pszModName* parameter does not specify a path and the filename extension, the **DosGetModHandle** function appends the default extension (*.dll*) and searches for the dynamic-link module in the directories specified by the **libpath** command in the *config.sys* file.

■ DosGetResource — Change

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

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

This function is included in MS OS/2 version 1.2 for compatibility purposes only. All new applications should use the **DosGetResource2** function, which returns a far pointer to the resource, rather than a selector.

Parameters

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

idType Specifies the type of resource to retrieve.

idName Specifies the name of the resource to retrieve.

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

Return Value

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

ERROR_CANT_FIND_RESOURCE
ERROR_INVALID_MODULE
ERROR_INVALID_SELECTOR

Comments

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

Type	Meaning
RT_ACCELTABLE	Accelerator tables
RT_BITMAP	Bitmap
RT_CHARTBL	Glyph-to-character tables
RT_DIALOG	Dialog template
RT_DISPLAYINFO	Screen-display information
RT_DLGINCLUDE	Dialog include file.
RT_FONT	Font
RT_FONTDIR	Font directory

Type	Meaning
RT_HELPSUBTABLE	Help-subtable resource.
RT_HELPTABLE	Help-table resource.
RT_KEYTBL	Key to UGL tables
RT_MENU	Menu template
RT_MESSAGE	Error-message tables
RT_POINTER	Mouse-pointer shape
RT_RCDATA	Binary data
RT_STRING	String tables
RT_VKEYTBL	Key to virtual-key tables

See Also **DosGetResource2, DosLoadModule**

Changes This function is included in MS OS/2, version 1.2, for compatibility purposes only. All new applications should use **DosGetResource2**.

DosGetResource2 New

```
USHORT DosGetResource2( hmod, idType, idName, ppData )
HMODULE hmod;          /* module handle                         */
USHORT  idType;        /* resource-type identifier              */
USHORT  idName;        /* resource-name identifier              */
PVOID FAR * ppData;    /* pointer to variable for resource address */
```

The **DosGetResource2** function retrieves a pointer to a resource.

Parameters

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

idType Specifies the type of resource to retrieve.

idName Specifies the name of the resource to retrieve.

ppData Points to the variable that receives the pointer to the resource.

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

ERROR_INVALID_PARAMETER
ERROR_INVALID_MODULETYPE

Comments The **DosGetResource2** function allocates a segment, copies the resource into the segment, and returns a pointer to the resource. A process can use this pointer to access the resource directly. For compatibility with future versions of MS OS/2, this function should be used instead of the **DosGetResource** function, which returns a selector instead of a pointer.

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

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

Example

This example calls **DosGetResource2** to retrieve a resource from the application's module, and then the calls **DosFreeResource** to free the memory used by the resource:

```
PBYTE pResource;

if (!DosGetResource2(NULL,  /* loads from application's module */
        RT_MENU,            /* gets a menu resource            */
        IDM_MENU,           /* ID of the menu to get           */
        &pResource)) {      /* pointer address                 */
    .
    .
    .
    DosFreeResource(pResource);  /* frees resource             */
```

See Also

DosFreeResource, DosGetResource, DosLoadModule

■ DosGetVersion — Correction

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

The **DosGetVersion** function retrieves version number of the operating system. For MS OS/2, version 1.1, both the major and minor version numbers are 10. For MS OS/2, version 1.2, the minor version number is 20.

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

Parameters

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

Return Value

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

Example

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

```
USHORT usVersion;
CHAR ch;

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

See Also

DosQSysInfo

Corrections

The example incorrectly retrieved the minor version number, instead of the major version number. It has been changed to show how to get and display both major and minor version numbers.

▮ DosLoadModule — Correction

USHORT DosLoadModule (*pszFailName*, *cbFileName*, *pszModName*, *phmod*)

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

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

Parameters

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

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

pszModName Points to the null-terminated string that specifies the module name. This string must be a valid MS OS/2 filename. If it does not specify a path and the filename extension, the function appends the default extension (*.dll*) and searches for the dynamic-link module in the directories specified by the **libpath** command in the *config.sys* file.

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

Return Value

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

ERROR_BAD_FORMAT
ERROR_FILE_NOT_FOUND
ERROR_INTERRUPT
ERROR_INVALID_NAME
ERROR_NOT_ENOUGH_MEMORY

Comments

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

Example

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

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

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

See Also

DosExecPgm, DosFreeModule, DosGetModName, DosGetProcAddr

Corrections

If the *pszModName* parameter does not specify a path and the filename extension, **DosLoadModule** function appends the default extension (*.dll*) and searches for the dynamic-link module in the directories specified by the **libpath** command in the *config.sys* file.

■ DosMakePipe — Change

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

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

Parameters

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

cbPipe Specifies the size (in bytes) to allocate for the storage buffer for this pipe. This can be any value up to 65,536 minus the size of the pipe header, which is currently 32 bytes. If this parameter is zero, the default buffer size is used. (The buffer size is advisory only. MS OS/2 may allocate more or less space.)

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

ERROR_NOT_ENOUGH_MEMORY
ERROR_TOO_MANY_OPEN_FILES

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

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

MS OS/2 temporarily blocks any call to the **DosWrite** function that would 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**

Changes The *cbPipe* parameter is advisory only. The actual buffer space allocated by the system may be larger (to a maximum of 65,536 minus the pipe header size) or smaller.

DosMkDir2 New

USHORT DosMkDir2(*pszDir, peaop, ulReserved* **)**

PSZ *pszDir*;	/* pointer to directory name */
PEAOP *peaop*;	/* pointer to structure for extended attributes */
ULONG *ulReserved*;	/* must be zero */

The **DosMkDir2** function creates a directory.

Parameters *pszDir* Points to a null-terminated string that specifies a valid MS OS/2 directory name.

peaop Points to the **EAOP** structure that defines extended attributes for the directory.

The **EAOP** structure has the following form:

```
typedef struct _EAOP {
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG    oError;
} EAOP;
```

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

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_EA_LIST_INCONSISTENT
ERROR_FILENAME_EXCED_RANGE
ERROR_INVALID_EA_NAME
ERROR_PATH_NOT_FOUND

Comments

Prior to the function call, the **fpFEAList** field of the **EAOP** structure should be set to point to the buffer that contains the relevant list of extended attributes.

If the *peaop* parameter is NULL, no extended attributes are defined for the directory.

If an error occurs during the creation of the extended attributes, the **oError** field of the **EAOP** structure will contain the offset within the list where the error occurred.

See Also

DosMkDir

■ DosMonReg — Change

```
USHORT DosMonReg(hmon, pbInBuf, pbOutBuf, fPosition, usIndex)
HMONITOR hmon;      /* monitor handle to register             */
PBYTE pbInBuf;      /* pointer to structure for input buffer  */
PBYTE pbOutBuf;     /* pointer to structure for output buffer */
USHORT fPosition;   /* position flag                          */
USHORT usIndex;     /* index                                  */
```

The **DosMonReg** function registers a monitor by placing it in a chain of other monitors for the same device. Each monitor receives input from or sends output to the device in the order in which it appears in the chain.

Parameters

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

pbInBuf Points to the **MONIN** structure that receives data from the device driver or from the previous monitor in the chain. The **MONIN** structure has the following form:

```
typedef struct _MONIN {
    USHORT cb;
    BYTE abReserved[18];
    BYTE bBuffer[108];
} MONIN;
```

pbOutBuf Points to the **MONOUT** structure that receives data for the next monitor in the chain. The **MONOUT** structure has the following form:

```
typedef struct _MONOUT {
    USHORT cb;
    BYTE abReserved[18];
    BYTE abBuffer[108];
} MONOUT;
```

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

Value	Meaning
MONITOR_BEGIN	Place the monitor at the beginning of the chain, ahead of any other monitors in the chain.
MONITOR_DEFAULT	Place the monitor anywhere in the chain.
MONITOR_END	Place the monitor at the end of the chain.

Any of the *fPosition* values may be combined with MONITOR_SPECIAL by using the OR operator to allow the monitor to continue to receive data even if the device is disabled or another monitor further down the chain is blocked. If the MONITOR_SPECIAL constant is not set, no monitors will receive input when the device driver is disabled or any monitor is blocked.

usIndex Specifies a device-specific value. If the device is the keyboard, *usIndex* specifies the ID for the screen group to monitor. If no screen-group number is available (the monitor is detached), the ID of the current foreground screen group can be obtained by calling **DosGetInfoSeg.** (The current foreground screen group is the group that most recently called **KbdCharIn.**)

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

ERROR_MON_BUFFER_TOO_SMALL
ERROR_MON_INVALID_HANDLE
ERROR_MON_INVALID_PARMS
ERROR_NOT_ENOUGH_MEMORY

Comments The **MONIN** and **MONOUT** structures must be in the same segment.

See Also **DosGetInfoSeg, DosMonClose, DosMonOpen, DosMonRead, DosMonWrite, KbdCharIn**

Changes A new value, MONITOR_SPECIAL, can be combined with any other position value for the *fPosition* parameter. This constant lets a monitor receive input even if the device is disabled or another monitor in the chain is blocked.

■ DosOpen Change

```
USHORT DosOpen(pszFileName, phf, pusAction, ulFileSize, usAttribute, fsOpenFlags, fsOpenMode,
                   ulReserved)
PSZ pszFileName;          /* pointer to filename                         */
PHFILE phf;               /* pointer to variable for file handle         */
PUSHORT pusAction;        /* pointer to variable for action taken        */
ULONG ulFileSize;         /* file size if created or truncated           */
USHORT usAttribute;       /* file attribute                              */
USHORT fsOpenFlags;       /* action taken if file exists/does not exist  */
USHORT fsOpenMode;        /* open mode of file                           */
ULONG ulReserved;         /* must be zero                                */
```

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

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

Parameters

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

phf Points to the variable that receives the handle of the opened file.

pusAction Points to the variable receiving the value that specifies the action taken by the **DosOpen** function. If **DosOpen** fails, this value has no meaning. Otherwise, it is one of the following values:

Value	Meaning
FILE_CREATED	File was created.
FILE_EXISTED	File already existed.
FILE_TRUNCATED	File existed and was truncated.

ulFileSize Specifies the file's new size (in bytes). This parameter applies only if the file is created or truncated. The size specification has no effect on a file that is opened only for reading.

usAttribute Specifies the file attributes. This parameter can be a combination of the following values:

Value	Meaning
FILE_NORMAL	File can be read from or written to.
FILE_READONLY	File can be read from, but not written to.
FILE_HIDDEN	File is hidden and does not appear in a directory listing.
FILE_SYSTEM	File is a system file.
FILE_ARCHIVED	File has been archived.

File attributes apply only if the file is created.

fsOpenFlags Specifies the action to take both when the file exists and when it does not exist. This parameter may be one of the following values:

Value	Meaning
FILE_CREATE	Create a new file; fail if the file already exists.
FILE_OPEN	Open an existing file; fail if the file does not exist.
FILE_OPEN | FILE_CREATE	Open an existing file or create the file if it does not exist.
FILE_TRUNCATE	Open an existing file and change to a given size.
FILE_TRUNCATE | FILE_CREATE	Open an existing file and truncate it, or create the file if it does not exist.

fsOpenMode Specifies the modes with which to open the file. It consists of one access mode and one share mode. The other values are optional and can be given in any combination:

Value	Meaning
OPEN_ACCESS_READONLY	Data can be read from the file but not written to it.
OPEN_ACCESS_READWRITE	Data can be read from or written to the file.
OPEN_ACCESS_WRITEONLY	Data can be written to the file but not read from it.
OPEN_SHARE_DENYNONE	Other processes can open the file for any access: read-only, write-only, or read-write.
OPEN_SHARE_DENYREAD	Other processes can open the file for write-only access but they cannot open it for read-only or read-write access.
OPEN_SHARE_DENYREADWRITE	The current process has exclusive access to the file. The file cannot be opened by any process (including the current process).
OPEN_SHARE_DENYWRITE	Other processes can open the file for read-only access but they cannot open it for write-only or read-write access.
OPEN_FLAGS_DASD	The file handle represents a physical drive that has been opened for direct access. (The *pszFileName* parameter must specify a drive name.) The **DosDevIOCtl** function can be used with this file handle to bypass the file system and to access the sectors of the drive directly.
OPEN_FLAGS_FAIL_ON_ERROR	Any function that uses the file handle returns immediately with an error value if there is an I/O error—for example, when the drive door is open or a sector is missing. If this value is not specified, the system passes the error to the system critical-error handler, which then reports the error to the user with a hard-error popup. The fail-on-error flag is not inherited by child processes. The fail-on-error flag applies to all functions that use the file handle, with the exception of the **DosDevIOCtl** function.

Value	Meaning
OPEN_FLAGS_NOINHERIT	The file handle is not available to any child process started by the current process. If this value is not specified, any child process started by the current process may use the file handle.
OPEN_FLAGS_WRITE_THROUGH	This flag applies to functions, such as **DosWrite**, that write data to the file. If this value is specified, the system writes data to the device before the given function returns. Otherwise, the system may store the data in an internal file buffer and write the data to the device only when the buffer is full or the file is closed.
OPEN_FLAGS_NO_LOCALITY	There is no specific information regarding the locality of reference (the degree of randomness with which the file is accessed).
OPEN_FLAGS_SEQUENTIAL	The file is accessed sequentially.
OPEN_FLAGS_RANDOM	The file is accessed randomly.
OPEN_FLAGS_RANDOMSEQUENTIAL	The file is accessed randomly, but that there is a degree of sequential I/O within that random access. For example, this flag is specified if large blocks of data are to be read or written at random locations in the file.
OPEN_FLAGS_NO_CACHE	The disk drive should not cache data in I/O operations on this file.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_CANNOT_MAKE
ERROR_DISK_FULL
ERROR_DRIVE_LOCKED
ERROR_FILE_NOT_FOUND
ERROR_INVALID_ACCESS
ERROR_INVALID_PARAMETER
ERROR_NOT_DOS_DISK
ERROR_OPEN_FAILED
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION
ERROR_TOO_MANY_OPEN_FILES

Comments

The ERROR_ACCESS_DENIED value is returned if you try to open a file in a mode that is incompatible with the file's current access and sharing modes—for example, if you attempt to open a read-only file for writing.

The ERROR_SHARING_VIOLATION value is returned if some other process has opened the file with a sharing method that denies the type of access you have requested.

Once the file is opened, the **DosSetFHandState** function can be used to change the OPEN_FLAGS_FAIL_ON_ERROR, OPEN_FLAGS_NOINHERIT, and OPEN_FLAGS_WRITE_THROUGH flags specified in *fsOpenMode*.

MS OS/2 does not provide a built-in method to inform a child process that it has inherited a given file handle. The parent process must pass this information to a child process. If the file is created without the OPEN_FLAGS_NOINHERIT flag, and the parent process terminates without closing the file, the file will remain open until all child processes have terminated.

Restrictions

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

- Only the access modes and the OPEN_FLAGS_DASD flag can be specified for the *fsOpenMode* parameter.

Example

This example calls the **DosOpen** function to create a file *abc* that is 100 bytes long and open it for write-only access. The *fsOpenFlags* parameter is set to FILE_CREATE so that **DosOpen** will return an error if the file already exists.

```
HFILE hf;
USHORT usAction;
DosOpen("abc",                                   /* filename to open       */
    &hf,                                         /* address of file handle */
    &usAction,                                   /* action taken           */
    100L,                                        /* size of new file       */
    FILE_NORMAL,                                 /* file attribute         */
    FILE_CREATE,                                 /* create the file        */
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, /* open mode              */
    0L);                                         /* reserved               */
```

See Also

DosBufReset, DosClose, DosDevIOCtl, DosDupHandle, DosQFHandState, DosQFileInfo, DosQFileMode, DosQFSInfo, DosSetFHandState, DosSet-FileMode, DosWrite

Changes

The following constants are new for the *fsOpenMode* parameter:

Value	Meaning
OPEN_FLAGS_NO_LOCALITY	There is no specific information regarding the locality of reference (the degree of randomness with which the file is accessed).
OPEN_FLAGS_SEQUENTIAL	The file is accessed sequentially.
OPEN_FLAGS_RANDOM	The file is accessed randomly.
OPEN_FLAGS_RANDOMSEQUENTIAL	The file is accessed randomly, but that there is a degree of sequential I/O within that random access. For example, this flag is specified if large blocks of data are to be read or written at random locations in the file.

Value	Meaning
OPEN_FLAGS_NO_CACHE	The disk drive should not cache data in I/O operations on this file.

Corrections

The comments incorrectly stated that ERROR_ACCESS_DENIED would be returned if another process had previously opened the file in an incompatible mode. The correct error code is ERROR_SHARING_VIOLATION.

■ DosOpen2 New

```
USHORT DosOpen2( pszFileName, phfHand, pusAction, ulFileSize, usAttribute, usOpenFlags,
                 ulOpenMode, peaop, ulReserved)
PSZ pszFileName;         /* pointer to filename                          */
PHFILE phfHand;          /* pointer to variable for file handle          */
PUSHORT pusAction;       /* pointer to variable for action taken         */
ULONG ulFileSize;        /* file size if created or truncated            */
USHORT usAttribute;      /* file attribute                               */
USHORT usOpenFlags;      /* action if file exists/does not exist         */
ULONG ulOpenMode;        /* open mode of file                            */
PEAOP peaop;             /* pointer to structure for extended attributes */
ULONG ulReserved;        /* must be zero                                 */
```

The **DosOpen2** function opens an existing file or creates a new file. This function returns a handle that can be used to read from and write to the file, as well as to retrieve information about the file.

For compatibility with future versions of MS OS/2, the **DosOpen2** function should be used instead of the **DosOpen** function.

Parameters

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

phfHand Points to the variable that receives the handle of the opened file.

pusAction Points to the variable receiving the value that specifies the action taken by the **DosOpen2** function. If **DosOpen2** fails, this value has no meaning. Otherwise, it is one of the following values:

Value	Meaning
FILE_CREATED	File was created.
FILE_EXISTED	File already existed.
FILE_TRUNCATED	File existed and was truncated.

ulFileSize Specifies the file's new size (in bytes). The size specification has no effect on a file that is opened only for reading.

usAttribute Specifies the file attributes. This parameter can be a combination of the following values:

Value	Meaning
FILE_NORMAL	File can be read from or written to.
FILE_READONLY	File can be read from, but not written to.

Value	Meaning
FILE_HIDDEN	File is hidden and does not appear in a directory listing.
FILE_SYSTEM	File is a system file.
FILE_ARCHIVED	File has been archived.

File attributes apply only if the file is created.

usOpenFlags Specifies the action to take both when the file exists and when it does not exist. This parameter can be one of the following values:

Value	Meaning
FILE_CREATE	Create a new file; fail if the file already exists.
FILE_OPEN	Open an existing file; fail if the file does not exist.
FILE_OPEN | FILE_CREATE	Open an existing file or create the file if it does not exist.
FILE_TRUNCATE	Open an existing file and change its size to a given size.
FILE_TRUNCATE | FILE_CREATE	Open an existing file and truncate it, or create the file if it does not exist.

ulOpenMode Specifies the modes with which to open the file. This parameter consists of one access mode and one share mode. All other values are optional; one locality mode can be specified, and the others can be given in any combination:

Value	Meaning
OPEN_ACCESS_READONLY	Data can be read from the file but not written to it.
OPEN_ACCESS_READWRITE	Data can be read from or written to the file.
OPEN_ACCESS_WRITEONLY	Data can be written to the file but not read from it.
OPEN_SHARE_DENYNONE	Other processes can open the file for any access: read-only, write-only, or read-write.
OPEN_SHARE_DENYREAD	Other processes can open the file for write-only access but they cannot open it for read-only or read-write access.
OPEN_SHARE_DENYREADWRITE	The current process has exclusive access to the file. No process (including the current process) can be open the file.
OPEN_SHARE_DENYWRITE	Other processes can open the file for read-only access but cannot open it for write-only or read-write access.

Value	Meaning
OPEN_FLAGS_DASD	The file handle represents a physical drive that has been opened for direct access. (The *pszFileName* parameter must specify a drive name.) The **DosDevIOCtl** function can be used with this file handle to bypass the file system and to access the sectors of the drive directly.
OPEN_FLAGS_FAIL_ON_ERROR	Any function that uses the file handle returns immediately with an error value if there is an I/O error—for example, when the drive door is open or a sector is missing. If this value is not specified, the system passes the error to the system critical-error handler, which then reports the error to the user with a hard-error popup. The fail-on-error flag is not inherited by child processes. The fail-on-error flag applies to all functions that use the file handle, with the exception of the **DosDevIOCtl** function.
OPEN_FLAGS_NOINHERIT	The file handle is not available to any child process started by the current process. If this value is not specified, any child process started by the current process can use the file handle.
OPEN_FLAGS_WRITE_THROUGH	This flag applies to functions (for example, **DosWrite**) that write data to the file. If this value is specified, the system writes data to the device before the given function returns. Otherwise, the system can store the data in a buffer and write the data to the device only when the buffer is full or the file is closed.
OPEN_FLAGS_NO_LOCALITY	There is no specific information regarding the locality of reference (the degree of randomness with which the file is accessed).
OPEN_FLAGS_SEQUENTIAL	The file is accessed sequentially.
OPEN_FLAGS_RANDOM	The file is accessed randomly.

Value	Meaning
OPEN_FLAGS_RANDOMSEQUENTIAL	The file is accessed randomly, but that there is a degree of sequential I/O within that random access. For example, this flag would be specified if large blocks of data were to be read or written at random locations in the file.
OPEN_FLAGS_NO_CACHE	The disk driver should not cache data in I/O operations on this file.

peaop Points to an **EAOP** structure that defines extended attributes for the file. If this value is NULL, the file will not use extended attributes. Before you call the **DosOpen2** function, the **fpFEAList** field of the **EAOP** structure must point to a data area where the relevant extended-attribute information is stored. The **EAOP** structure has the following form:

```
typedef struct _EAOP {
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG    oError;
} EAOP;
```

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

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_DISK_FULL
ERROR_EA_LIST_INCONSISTENT
ERROR_EA_VALUE_UNSUPPORTABLE
ERROR_FILE_NOT_FOUND
ERROR_FILENAME_EXCED_RANGE
ERROR_INVALID_ACCESS
ERROR_INVALID_EA_NAME
ERROR_INVALID_PARAMETER
ERROR_OPEN_FAILED
ERROR_PATH_NOT_FOUND
ERROR_SHARING_BUFFER_EXCEEDED
ERROR_SHARING_VIOLATION
ERROR_TOO_MANY_OPEN_FILES

Comments

The read/write pointer is initially set at the first byte of the file.

The *ulFileSize* parameter affects the size of the file only when it is created, truncated, or replaced. The value specified for this parameter is the recommended file size. The file can be opened even if allocation of the full amount of bytes fails.

The value of the *usOpenFlags* parameter provides a disk-access mechanism that is independent of the file system. When this value is used, the **DosOpen2** function returns a handle to the calling process that represents the physical drive as a file. In order to prevent other processes from accessing the disk, the calling process must also issue a **DosDevIOCtl** DSK_LOCKDRIVE subcall, which requires the file handle returned by the **DosOpen2** function for the physical drive.

Extended attributes that require contiguous disk space may cause the function to fail if the file system is unable to allocate contiguous space.

DosOpen2 sets extended attributes when a file is created, replaced, or truncated. Extended attributes are ordinarily set when a file is opened for reading. When a file is replaced, the extended attributes are also replaced. Extended attributes are discarded if the *peaop* parameter is NULL.

The *pszFileName* parameter cannot point to a volume label, because volume labels cannot be opened.

Any sharing restrictions placed on a file when it is opened are removed when it is closed. When a file is inherited by a child process, all sharing and access restrictions are also inherited.

The **DosOpen2** function opens the client end of a named pipe and returns a handle of the pipe. The pipe must be in "listen" state for the open operation to succeed; otherwise the open operation fails and the ERROR_PIPE_BUSY error value is returned. Until a given instance of a named pipe has been closed by a client, that same instance cannot be opened by another client; however, the opening process can duplicate the open handle as many times as required. The access and sharing modes specified when a pipe is opened must be consistent with the modes specified in the call to the **DosMakeNmPipe** function. Pipes are always opened with the pipe-specific states set to lock read and write operations and are read as a byte stream.

See Also **DosClose, DosDevIOCtl, DosDupHandle, DosMakeNmPipe, DosOpen, DosSetFHandState, DosSetFileInfo**

■ DosQFHandState — Change

```
USHORT DosQFHandState(hf, pfsOpenMode)
HFILE hf;                 /* file handle                           */
PUSHORT pfsOpenMode;      /* pointer to variable for file-handle state */
```

The **DosQFHandState** function retrieves the state of the specified file handle. The file-handle state indicates whether the file may be read from or written to and whether it may be opened for reading or writing by other processes.

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

Parameters

hf Identifies the file whose file-handle state is to be retrieved. This handle must have been previously created by using the **DosOpen** function.

pfsOpenMode Points to the variable that receives the file-handle state. The file-handle state consists of one access mode, one share mode, and optional flags. It is identical to the values specified in the *fsOpenMode* parameter of the **DosOpen** function. Which values are set can be determined by using the AND operator to combine the value returned in the *pfsOpenMode* parameter with one or more of the following values:

Value	Meaning
OPEN_ACCESS_READONLY	Data can be read from the file but not written to it.
OPEN_ACCESS_READWRITE	Data can be read from or written to the file.

Value	Meaning
OPEN_ACCESS_WRITEONLY	Data can be written to the file but not read from it.
OPEN_SHARE_DENYNONE	Other processes can open the file for any access: read-only, write-only, or read-write.
OPEN_SHARE_DENYREAD	Other processes can open the file for write-only access but not for read-only or read-write access.
OPEN_SHARE_DENYREADWRITE	The current process has exclusive access to the file.
OPEN_SHARE_DENYWRITE	Other processes can open the file for read-only access but not for write-only or read-write access.
OPEN_FLAGS_DASD	The file handle represents a physical drive that has been opened for direct access.
OPEN_FLAGS_FAIL_ON_ERROR	Any function that uses the file handle returns immediately with an error code if there is an I/O error.
OPEN_FLAGS_NOINHERIT	The file handle is private to the current process.
OPEN_FLAGS_WRITE_THROUGH	The system writes data to the device before the given function returns.
OPEN_FLAGS_NO_CACHE	The system does not cache file data.

Return Value

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

ERROR_INVALID_HANDLE

Example

This example calls the **DosQFHandState** function using the handle of a previously opened file, and then checks the variable fsOpenMode and reports if the file is opened for read/write access:

```
HFILE hf;
USHORT fsOpenMode;
    .
    .
    .
DosQFHandState(hf, &fsOpenMode);
if (fsOpenMode & OPEN_ACCESS_READWRITE)
    VioWrtTTY("File opened for read/write access\r\n", 35, 0);
if (fsOpenMode & OPEN_SHARE_DENYREADWRITE)
    VioWrtTTY("File cannot be shared\r\n", 23, 0);
```

See Also

DosDevIOCtl, DosExecPgm, DosOpen, DosSetFHandState

Changes

The OPEN_FLAGS_NO_CACHE value can be specified for the *pfsOpenMode* parameter. If specified, the system does not cache file data.

■ DosQFileInfo Change

```
USHORT DosQFileInfo( hf, usInfoLevel, pvInfo, cbInfoBuf)
HFILE hf;              /* handle of file about which data sought */
USHORT usInfoLevel;    /* level of file data required            */
PVOID pvInfo;          /* pointer to file-data buffer            */
USHORT cbInfoBuf;      /* length of file-data buffer             */
```

The **DosQFileInfo** function retrieves information about a specific file. The file information consists of the date and time the file was created, the date and time it was last accessed, the date and time it was last written to, the size of the file, and its attributes. It can also be used to return information about the extended attributes used for a file.

The file information is based on the most recent call to the **DosClose** or the **DosBufReset** function.

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

Parameters

hf Identifies the file about which information is to be retrieved. This handle must have been created by using the **DosOpen** function.

usInfoLevel Specifies the level of file information required. It may be one of the following values:

Value	Meaning
FILE_INFO_1	Level-1 information request. This will return a **FILESTATUS** structure. Any time and data fields in the structure that the file-system device does not support are set to zero.
FILE_INFO_2	Level-2 information request. This will return a **FILESTATUS2** structure, which contains the same information as **FILESTATUS** plus the size of the structure used by the FILE_INFO_3 value (for MS OS/2 version 1.2, this size cannot exceed 65,535 bytes).
FILE_INFO_3	Level-3 information request. This will return an **EAOP** structure that contains a subset of the file's extended-attribute information.

pvInfo Points to the structure that receives the file information. This structure will be **FILESTATUS** for FILE_INFO_1 information, **FILESTATUS2** for FILE_INFO_2 information, and **EAOP** for FILE_INFO_3 information.

The **FILESTATUS** structure has the following form:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

The **FILESTATUS2** structure has the following form:

```
typedef struct _FILESTATUS2 {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
    USHORT cbList;
} FILESTATUS2;
```

The **EAOP** structure has the following form:

```
typedef struct _EAOP {
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG    oError;
} EAOP;
```

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

cbInfoBuf Specifies the length (in bytes) of the buffer that receives the file information.

Return Value

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

ERROR_INVALID_EA_NAME
ERROR_EA_LIST_INCONSISTENT
ERROR_BUFFER_OVERFLOW
ERROR_DIRECT_ACCESS_HANDLE
ERROR_INVALID_HANDLE
ERROR_INVALID_LEVEL

Comments

Prior to the function being called, the **fpFEAlist** field in the **EAOP** structure should be initialized so that it points to the **FEALIST** structure that contains the relevant **FEA** structure. The **cbList** field in the **FEALIST** structure is valid, giving the size of the **FEA** structure.

If the **FEALIST** structure is not large enough to hold the returned information (indicated by ERROR_BUFFER_OVERFLOW), **cbList** will still be valid, assuming there is at least enough space for it. Its value will be the size of the entire set of extended attributes for the file, even if only a subset of attributes was requested.

Example

This example opens the file *abc*, calls the **DosQFileInfo** function to retrieve the current allocated size, and then calls the **DosNewSize** function to increase the file's size by 1K:

```
HFILE hf;
USHORT usAction;
FILESTATUS fstsFile;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL,
    FILE_OPEN | FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, 0L);
DosQFileInfo(hf,                    /* file handle             */
    FILE_INFO_1,                    /* level of information    */
    &fstsFile,                      /* address of file-data buffer */
    sizeof(fstsFile));              /* size of data buffer     */
DosNewSize(hf, fstsFile.cbFileAlloc + 1024L);
```

See Also **DosBufReset, DosClose, DosNewSize, DosOpen, DosQFileMode, DosQPathInfo, DosSetFileInfo**

Changes Parameters and structures for FILE_INFO_2 and FILE_INFO_2 information have been added. The type of the *pvInfo* parameter has changed from PFILESTATUS to PVOID because one of three structures can be used for this parameter.

■ DosQFSAttach New

```
USHORT DosQFSAttach(pszDev, usOrdinal, usInfoLevel, pFSAttBuf, pcbAttBuf, ulReserved)
PSZ pszDev;            /* pointer to drive or device                         */
USHORT usOrdinal;      /* index to drive or device                           */
USHORT usInfoLevel;    /* level of information                               */
PBYTE pFSAttBuf;       /* pointer to structure for file-system attributes    */
PUSHORT pcbAttBuf;     /* pointer to structure length                        */
ULONG ulReserved;      /* must be zero                                       */
```

The **DosQFSAttach** function queries information about an attached remote file system or a local file system. The function can also query information about a character device or pseudo-character device attached to a local or remote file system.

Parameters

pszDev Points to a null-terminated string that specifies the drive letter followed by a colon or to the name of a character or pseudo-character device. If this parameter is a character or pseudo-character device name, the format of the string is \DEV\filename, where *filename* is a valid MS OS/2 filename. This parameter is ignored if the *usInfoLevel* parameter is set to either FSAIL_DEVNUMBER or FSAIL_DRVNUMBER.

usOrdinal Specifies an index into the list of character or pseudo-character devices or the set of drives. The first item in the list is always 1. This parameter is ignored if the *usInfoLevel* parameter is set to FSAIL_QUERYNAME.

usInfoLevel Specifies the type of information requested. This parameter can be one of the following values:

Value	Meaning
FSAIL_QUERYNAME	Returns information about the drive or device pointed to by the *pszDev* parameter. When this value is specified, the *usOrdinal* parameter is ignored.
FSAIL_DEVNUMBER	Returns information about the character or pseudo-character device specified by the *usOrdinal* parameter. When this value is specified, the *pszDev* parameter is ignored.
FSAIL_DRVNUMBER	Returns information about the drive specified by the *usOrdinal* parameter. When this value is specified, the *pszDev* parameter is ignored.

pFSAttBuf Points to the buffer that receives information about the file system. The buffer is organized as a **FSQBUFFER** structure. Because the name fields can vary length, however, the structure cannot be used directly to retrieve the data. The **PFSQBUFFER** structure has the following form:

```
typedef struct _FSQBUFFER {
    USHORT  iType;
    USHORT  cbName;
    UCHAR   szName[1];
    USHORT  cbFSDName;
    UCHAR   szFSDName[1];
    USHORT  cbFSAData;
    UCHAR   rgFSAData[1];
} FSQBUFFER;
```

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

pcbAttBuf Points to the variable that receives the length (in bytes) of the buffer.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_BUFFER_OVERFLOW
ERROR_INVALID_DRIVE
ERROR_INVALID_LEVEL
ERROR_NO_MORE_ITEMS

Comments

The **DosQFSAttach** function can be used to ensure that the correct file system is loaded for a disk. Without this information, there is potential for the data on the disk to be destroyed because the wrong file system could be attached to the disk by default.

Example

This example calls **DosQFSAttach** to get information about drive C, and then displays the device and file-system names:

```
PSZ psz;
PUSHORT pcb;
USHORT cb;
SEL sel;

DosAllocSeg(1024, &sel, SEG_NONSHARED); /* allocates buffer        */

if (!DosQFSAttach("c:", 0, FSAIL_QUERYNAME, MAKEP(sel, 0), &cb, 0L)) {
    pcb = MAKEP(sel, 2);          /* points to length of device name */
    psz = MAKEP(sel, 4);          /* points to device name           */
    VioWrtTTY(psz, *pcb, NULL); /* displays device name            */
    VioWrtTTY("\r\n", 2, 0L);
    psz += *pcb + 3;        /* add null char. and name-length var.   */
    pcb = (PUSHORT) (psz - 2);  /* points to length of name        */
    VioWrtTTY(psz, *pcb, NULL); /* displays file-system name       */
    VioWrtTTY("\r\n", 2, 0L);
}
```

See Also

DosFSAttach, DosQFSInfo

■ DosQNmPipeInfo — Change

```
USHORT DosQNmPipeInfo(hp, usInfoLevel, pbBuf, cbBuf)
HPIPE hp;              /* pipe handle                       */
USHORT usInfoLevel;    /* level of information to retrieve  */
PBYTE pbBuf;           /* pointer to buffer for information */
USHORT cbBuf;          /* number of bytes in buffer         */
```

The **DosQNmPipeInfo** function retrieves information about a named pipe.

Parameters

hp Identifies the pipe to read from.

usInfoLevel Specifies the level of information to retrieve. Level 1 is miscellaneous information about the pipe.

pbBuf Points to the buffer that receives the information. For level-1 information, the data is stored in the **PIPEINFO** structure. The **PIPEINFO** structure has the following form:

```
typedef struct _PIPEINFO {
    USHORT cbOut;
    USHORT cbIn;
    BYTE   cbMaxInst;
    BYTE   cbCurInst;
    BYTE   cbName;
    CHAR   szName[1];
} PIPEINFO;
```

cbBuf Specifies the size (in bytes) of the buffer receiving the information.

Return Value

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

ERROR_BAD_PIPE
ERROR_BUFFER_OVERFLOW
ERROR_INVALID_LEVEL
ERROR_INVALID_PARAMETER
ERROR_PIPE_NOT_CONNECTED

Comments

For level-1 information, if the pipe name is longer than 255 bytes, zero will be returned in the **cbName** field of the **PIPEINFO** structure. The full null-terminated string that contains the name will be returned in the location specified by the **szName** field.

See Also

DosQNmPHandState, DosQNmPipeSemState

Changes

Pipe names longer than 255 bytes are now supported. For names longer than 255 bytes, however, zero is returned in the **cbName** field of the **PIPEINFO** structure.

Corrections

This function returns only level-1 information. Erroneous references to level-2 information have been removed.

■ DosQNmPipeSemState — Correction

```
USHORT DosQNmPipeSemState( hsem, pnmpsmst, cbBuf)
HSEM hsem;                    /* semaphore handle                     */
PPIPESEMSTATE pnmpsmst;       /* pointer to buffer receiving information */
USHORT cbBuf;                 /* buffer size                          */
```

The **DosQNmPipeSemState** function returns information about all local named pipes that are in blocking mode and are associated with a specified system semaphore.

Parameters

hsem Identifies the semaphore that is associated with the named pipe.

pnmpsmst Points to the **PIPESEMSTATE** structure that receives the information. The **PIPESEMSTATE** structure has the following form:

```
typedef struct _PIPESEMSTATE {
    BYTE    fStatus;
    BYTE    fFlag;
    USHORT usKey;
    USHORT usAvail;
} PIPESEMSTATE;
```

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

cbBuf Specifies the length (in bytes) of the structure that receives the information. Programs written in the C language should use the sizeof operator to set this parameter.

Return Value

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

ERROR_INVALID_PARAMETER
ERROR_SEM_NOT_FOUND

See Also

DosSetNmPipeSem

Corrections

The second parameter has been replaced by a **PIPESEMSTATE** structure.

■ DosQPathInfo — New

```
USHORT DosQPathInfo( pszPath, usInfoLevel, pInfoBuf, cbInfoBuf, ulReserved)
PSZ pszPath;            /* pointer to path                  */
USHORT usInfoLevel;     /* level of information             */
PBYTE pInfoBuf;         /* pointer to buffer for information */
USHORT cbInfoBuf;       /* length of information buffer     */
ULONG ulReserved;       /* must be zero                     */
```

The **DosQPathInfo** function returns information about a specified file or directory.

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

Parameters

pszPath Points to the null-terminated string that specifies the path of the file or directory. Wildcard characters are valid in the path only when the value of the *usInfoLevel* parameter is FIL_QUERYFULLNAME or FIL_NAMEISVALID.

usInfoLevel Specifies the level of information required. This parameter can be one of the following values:

Value	Meaning
FIL_STANDARD	Return a **FILESTATUS** structure.
FIL_QUERYEASIZE	Return a **FILESTATUS** structure followed by a 4-byte value that specifies the buffer size needed to retrieve the entire extended attribute.
FIL_QUERYEASFROMLIST	Return extended-attribute information using an **EAOP** structure for the *pInfoBuf* parameter.
FIL_QUERYFULLNAME	Return the fully qualified path of the buffer pointed to by the *pInfoBuf* parameter. When this value is specified, the path pointed to by the *pszPath* parameter can contain wildcard characters.
FIL_NAMEISVALID	Verify the correctness (according to MS OS/2 syntax rules) of the path pointed to by the *pszPath* parameter. If the path is incorrect (for example, a filename is too long for the current file system), an error will be returned. The path can contain wildcard characters.

pInfoBuf Points to the buffer that contains a **FILESTATUS** or **EAOP** structure. The structure used is determined by the value specified for the *usInfoLevel* parameter.

The **FILESTATUS** structure has the following form:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

The **EAOP** structure has the following form:

```
typedef struct _EAOP {
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG    oError;
} EAOP;
```

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

cbInfoBuf Specifies the length (in bytes) of the buffer pointed to by the *pInfoBuf* parameter.

ulReserved Specifies a reserved value; must be zero.

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

ERROR_BUFFER_OVERFLOW
ERROR_EA_LIST_INCONSISTENT
ERROR_FILENAME_EXCED_RANGE
ERROR_INVALID_EA_NAME
ERROR_INVALID_LEVEL
ERROR_PATH_NOT_FOUND

Comments If the *usInfoLevel* parameter is FIL_QUERYEASFROMLIST, a subset of the extended-attribute information for the file is returned. Prior to the call to the **DosQPathInfo** function, the **fpGEAList** field of the **EAOP** structure should point to a list that defines the attribute names for which values will be returned, and the **fpFEAList** field should point to a buffer in which the relevant extended-attribute list will be returned.

See Also **DosQFileInfo, DosSetPathInfo**

■ DosRead — Correction

```
USHORT DosRead(hf, pvBuf, cbBuf, pcbBytesRead)
HFILE hf;                    /* file handle                              */
PVOID pvBuf;                 /* pointer to buffer receiving data         */
USHORT cbBuf;                /* number of bytes in buffer                */
PUSHORT pcbBytesRead;        /* pointer to variable for number of bytes read */
```

The **DosRead** function reads up to a specified number of bytes of data from a file into a buffer. The function may read fewer than the specified number of bytes if it reaches the end of the file.

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

Parameters *hf* Identifies the file to be read. This handle must have been created by using the **DosOpen** function.

pvBuf Points to the buffer that receives the data.

cbBuf Specifies the number of bytes to read from the file.

pcbBytesRead Points to the variable that receives the number of bytes read from the file. This parameter is zero if the file pointer is positioned at the end of the file prior to the call to the **DosRead** function.

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

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NOT_DOS_DISK

Comments The **DosRead** function does not return an error if the file pointer is at the end of the file when the read operation begins.

When **DosRead** is used to read a byte pipe, the pipe must be in byte-read mode, an error is returned if the pipe is in message-read mode. All currently available data, up to the size requested, is returned.

For a message pipe in message-read mode, a read operation that is larger than the next available message returns only that message, with *pcbBytesRead* set to indicate the size of the returned message. A read operation that is smaller than the next available message returns with the number of bytes requested and an ERROR_MORE_DATA error code. Subsequent **DosRead** calls will continue reading the message. The **DosPeekNmPipe** function can be used to determine how many bytes are left in the message.

For a message pipe in byte-read mode, **DosRead** reads the pipe as if it were a byte stream, skipping over message headers. This is the same as reading a byte pipe in byte mode.

When blocking mode is set, the read operation blocks until data is available. In this case, the read operation will never return with the *pcbBytesRead* parameter equal to zero except when it has read an end-of-file (EOF) character. Note that in message-read mode, messages are always read entirely, except in the case where the message is larger than the size specified for the read operation.

When nonblocking mode is set, the read operation returns with the *pcbBytesRead* parameter equal to zero upon reading the EOF character. An error will be returned if no data is available.

When resuming reading a message after an ERROR_MORE_DATA error occurs, the read operation always blocks until the next part of the message can be transferred. When nonblocking mode is set, the read operation can return with *pcbBytesRead* equal to zero if, upon attempting to read at the start of a message, it determines that no message is available.

Example

This example opens, reads, and displays the file *abc*:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesRead, cbBytesWritten;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
do {
    DosRead(hf,            /* file handle                       */
        abBuf,             /* address of buffer                 */
        sizeof(abBuf),     /* size of buffer                    */
        &cbBytesRead);     /* address for number of bytes read */
    DosWrite(1, abBuf, cbBytesRead, &cbBytesWritten);
}
while (cbBytesRead);
```

See Also

DosChgFilePtr, DosOpen, DosPeekNmPipe, DosReadAsync, DosWrite, KbdStringIn

Corrections

DosRead can be used to read from a named pipe. The comments have been updated to contain the relevant information about reading from a named pipe.

DosReadAsync — Change

```
USHORT DosReadAsync( hf, hsemRam, pusErrCode, pvBuf, cbBuf, pcbBytesRead)
HFILE hf;                /* file handle                                  */
PULONG hsemRam;          /* pointer to RAM semaphore                     */
PUSHORT pusErrCode;      /* pointer to variable for error return code    */
PVOID pvBuf;             /* pointer to input buffer                      */
USHORT cbBuf;            /* length of input buffer                       */
PUSHORT pcbBytesRead;    /* pointer to variable for number of bytes read */
```

The **DosReadAsync** function reads one or more bytes of data from the file identified by the *hf* parameter. The function reads the data asynchronously; that is, the function returns immediately to the process that called it but continues to copy data to the specified buffer while the process continues.

Parameters

hf Identifies the file to be read. This handle must have been previously opened by using the **DosOpen** function.

hsemRam Points to the RAM semaphore that indicates when the function has finished reading the data.

pusErrCode Points to the variable that receives any error code the function generates while reading data. The possible error codes are identical to those returned by the **DosRead** function.

pvBuf Points to the buffer that receives the data being read.

cbBuf Specifies the number of bytes to be read from the file identified by the *hf* parameter.

pcbBytesRead Points to the variable that receives the number of bytes read from the file.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NO_PROC_SLOTS
ERROR_NOT_DOS_DISK

Comments

The **DosReadAsync** function reads up to the number of bytes specified in the *cbBuf* parameter, but it may read fewer if it reaches the end of the file. In any case, the function copies the number of bytes read to the variable pointed to by the *pcbBytesRead* parameter. The *pcbBytesRead* parameter is zero if all the bytes in the file have been read (that is, the end of file has been reached).

If the process intends to use the RAM semaphore pointed to by the *hsemRam* parameter to determine when data is available, it must set the semaphore by using the **DosSemSet** function before calling **DosReadAsync**. When **Dos-ReadAsync** has read the data, it clears the RAM semaphore.

The **DosReadAsync** function carries out the asynchronous operation by creating a new thread that reads from the specified file. The function terminates the thread when the operation is complete or when an error occurs.

When **DosReadAsync** is used to read a byte pipe, the pipe must be in byte-read mode; an error is returned if the pipe is in message-read mode. All currently available data, up to the size requested, is returned.

For a message pipe in message-read mode, a read operation that is larger than the next available message returns only that message; *pcbBytesRead* is set to indicate the size of the message returned. A read operation that is smaller than the next available message returns with the number of bytes requested and an ERROR_MORE_DATA error code. Subsequent **DosReadAsync** calls will continue reading the message. **DosPeekNmPipe** may be used to determine how many bytes are left in the message.

For a message pipe in byte-read mode, **DosReadAsync** reads the pipe as if it were a byte stream, skipping over message headers. This is the same as reading a byte pipe in byte mode.

When blocking mode is set, a read operation blocks until data is available. In this case, the read operation will not return with the *pcbBytesRead* parameter equal to zero except when it has read an end-of-file (EOF) character. Note that in message-read mode, messages are always read entirely, except in the case where the message is larger than the size specified for the read operation.

When nonblocking mode is set, a read operation returns with *pcbBytesRead* equal to zero upon reading the EOF character. An error will be returned if there is no data available.

When resuming reading a message after an ERROR_MORE_DATA message, the read operation always blocks until the next part of the message can be transferred. When nonblocking mode is set, the read operation can return with *pcbByteRead* equal to zero if, upon attempting to read at the start of a message, it determines that no message is available.

Example

This example opens the file *abc*, sets a RAM semaphore, and calls the **DosReadAsync** function to read part of the file. While the file is being read, program execution continues until the call to the **DosSemWait** function, which does not return until the **DosReadAsync** thread completes its work.

```
BYTE abBuf[512];
ULONG hReadSemaphore = 0;
HFILE hf;
USHORT usAction, cbBytesRead;
USHORT usReadReturn;
DosOpen("abc", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
DosSemSet(&hReadSemaphore);      /* sets RAM semaphore          */
DosReadAsync(hf,                 /* handle to file              */
    &hReadSemaphore,             /* address of semaphore        */
    &usReadReturn,               /* address to store return code */
    abBuf,                       /* address of buffer           */
    sizeof(abBuf),               /* size of buffer              */
    &cbBytesRead);               /* number of bytes read        */
    .
    . /* Other processing occurs here. */
    .
DosSemWait(&hReadSemaphore, -1L);
```

See Also

DosOpen, DosPeekNmPipe, DosRead, DosSemSet, DosSemWait, DosWriteAsync

Changes

Information about using this function with pipes has been added.

DosReadQueue — Correction

```
USHORT DosReadQueue( hqueue, pqresc, pcbElement, ppv, usElement, fWait, pbElemPrty, hsem)
HQUEUE hqueue;               /* handle of queue to read                    */
PQUEUERESULT pqresc;         /* pointer to structure for PID and request code */
PUSHORT pcbElement;          /* pointer to variable for length of element  */
PVOID FAR * ppv;             /* pointer to buffer for element              */
USHORT usElement;            /* element number to read                     */
UCHAR fWait;                 /* wait/no wait indicator                     */
PBYTE pbElemPrty;            /* pointer to variable for priority of element */
HSEM hsem;                   /* semaphore handle                           */
```

The **DosReadQueue** function retrieves an element and then removes it from a queue. It copies the address of the element to the supplied pointer and fills a structure with information about the element.

Parameters

hqueue Identifies the queue to read. This handle must have been created or opened by using the **DosCreateQueue** or **DosOpenQueue** function.

pqresc Points to the **QUEUERESULT** structure that receives information about the request. The **QUEUERESULT** structure has the following form:

```
typedef struct _QUEUERESULT {
    PID pidProcess;
    USHORT usEventCode;
} QUEUERESULT;
```

pcbElement Points to the variable that receives the length (in bytes) of the element.

ppv Points to the pointer that receives the address of the element in the queue.

usElement Specifies where to look in the queue for the element. If this parameter is 0x0000, the function looks at the beginning of the queue. Otherwise, the function assumes the value is an element identifier retrieved by using the **DosPeekQueue** function and looks for the specified element.

fWait Specifies whether to wait for an element to be placed in the queue, if the queue is empty. If this parameter is DCWW_WAIT, the function waits until an element is available. If this parameter is DCWW_NOWAIT, the function returns immediately with a code that indicates there are no entries in the queue.

pbElemPrty Points to the variable that receives the priority value specified when the element was added to the queue. This is a value in the range 0 through 15; 15 indicates the highest priority.

hsem Identifies a semaphore. This value can be the handle of a system semaphore that has been created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore. This semaphore would typically be used in a call to the **DosMuxSemWait** function to wait until the queue has an element. If the *fWait* parameter is DCWW_WAIT, *hsem* is ignored.

Return Value

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

ERROR_QUE_ELEMENT_NOT_EXIST
ERROR_QUE_EMPTY
ERROR_QUE_INVALID_HANDLE
ERROR_QUE_INVALID_WAIT
ERROR_QUE_PROC_NOT_OWNED

Comments

If the queue is empty, the **DosReadQueue** function either returns immediately or waits for an element to be written to the queue, depending on the value of the *fWait* parameter.

Only the process that created the queue can call the **DosReadQueue** function.

Example

This example reads the queue and waits until an element is received. After the element is read and the data processed, the process frees the shared memory that was passed to it. This assumes the process writing to the queue created a shared-memory segment. For more information, see the **DosWriteQueue** function.

```
QUEUERESULT qresc;
USHORT cbElement;
PVOID pv;
BYTE bElemPrty;

DosReadQueue(hqueue,  /* queue handle                          */
    &qresc,           /* address of result structure           */
    &cbElement,       /* receives element number               */
    &pv,              /* receives data address                 */
    0,                /* element number to read                */
    DCWW_WAIT,        /* waits until something is written      */
    &bElemPrty,       /* receives priority level               */
    NULL);            /* semaphore not needed, since waiting */
    .
    . /* Process the data. */
    .
DosFreeSeg(SELECTOROF(pv));     /* frees shared memory */
```

See Also

DosCreateQueue, DosMuxSemWait, DosOpenQueue, DosOpenSem, DosPeekQueue, DosWriteQueue

Corrections

The description incorrectly stated that the element was copied to the supplied buffer. It is the address of the element that is copied to the supplied pointer. No data is actually copied; only the pointer to the data is copied.

■ DosReallocHuge — Change

USHORT DosReallocHuge(*usNumSeg*, *usPartialSeg*, *sel* **)**

```
USHORT usNumSeg;       /* number of 65,536-byte segments */
USHORT usPartialSeg;   /* number of bytes in last segment */
SEL sel;               /* segment selector                */
```

The **DosReallocHuge** function reallocates a huge-memory block. This function changes the size of the huge memory to the specified number of 65,536-byte segments plus an additional segment of a specified size.

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

Parameters

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

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

sel Specifies the selector for the huge-memory block to be reallocated. The selector must have been created by using the **DosAllocHuge** function.

Return Value

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

ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosReallocHuge** function does not change the sharable and discardable attributes of the segments in the huge-memory block. If it was originally a sharable or discardable block, it remains a sharable or discardable block. However, if **DosReallocHuge** reallocates a discardable block, it also locks the segments. The **DosUnlockSeg** function must be used to unlock the segments and permit discarding.

The huge-memory block cannot be reallocated for a size larger than the maximum specified by the *usMaxNumSeg* parameter in the original call to the **DosAllocHuge** function.

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

Typically, **DosReallocHuge** can increase, not decrease, the size of shared huge segments. If the shared segment is allocated by the **DosAllocHuge** function, the segment can be decreased in size by setting the *fsAttr* parameter to SEG_SIZEABLE.

DosReallocHuge can be issued from ring 2, but only ring-3 segments are affected by this function.

Restrictions

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

- The *usPartialSeg* parameter is rounded up to the next paragraph (16-byte) value.

See Also

DosAllocHuge, DosFreeSeg, DosGetHugeShift, DosLockSeg, DosReallocSeg, DosUnlockSeg

Changes

Typically, **DosReallocHuge** can increase, not decrease, the size of shared huge segments. If the shared segment is allocated by the **DosAllocHuge** function, the segment can be decreased in size by setting the *fsAttr* parameter to SEG_SIZEABLE.

DosReallocHuge can be issued from ring 2, but only ring-3 segments are affected by this function.

■ DosReallocSeg — Change

```
USHORT DosReallocSeg( usNewSize, sel)
USHORT usNewSize;     /* new segment size */
SEL sel;              /* segment selector */
```

The **DosReallocSeg** function reallocates a segment. The function changes the size of the segment to the number of bytes specified by the *usNewSize* parameter.

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

Parameters

usNewSize Specifies the new size (in bytes). The size can be any number from 0 through 65,535. If it is 0, the function allocates 65,536 bytes.

sel Specifies the selector of the segment to be reallocated. The selector must have been created previously by using **DosAllocSeg** or **DosAllocShrSeg**.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_NOT_ENOUGH_MEMORY

Comments

The **DosReallocSeg** function does not change the sharable and discardable attributes of the segment. If it was originally a sharable or discardable segment, it remains a sharable or discardable segment. If **DosReallocSeg** reallocates a discardable segment, however, it also locks the segment. You must use the **DosUnlockSeg** function to unlock the segment and permit discarding.

If the **DosReallocSeg** function is used to reallocate a shared segment to a size smaller than its original size, the segment must have been created using the **DosAllocSeg** function with the SEG_SIZEABLE attribute set. This request can be issued from ring 2 or ring 3; the segment to be reallocated can be a ring-2 or a ring-3 segment.

The **DosReallocSeg** function cannot be used to reallocate a segment created by the **DosCreateCSAlias** function.

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.

Example

This example allocates a segment with 16,000 bytes, and then calls **DosReallocSeg** to increase the size to 32,000 bytes:

```
SEL sel;

DosAllocSeg(16000, &sel, SEG_NONSHARED);    /* allocates memory   */
      .
      .
      .
DosReallocSeg(32000, sel);                  /* reallocates memory */
```

See Also

DosAllocSeg, DosFreeSeg, DosLockSeg, DosReallocHuge, DosUnlockSeg

Changes

If **DosReallocSeg** is used to reallocate a shared segment to a size smaller than its original size, the segment must have been created using the **DosAllocSeg** function with the SEG_SIZEABLE attribute set. This request can be issued from ring 2 or ring 3; the segment to be reallocated can be either a ring-2 or a ring-3 segment.

■ DosSearchPath Change

```
USHORT DosSearchPath( fsSearch, pszPath, pszFileName, pbBuf, cbBuf)
USHORT fsSearch;     /* search flags                                    */
PSZ pszPath;         /* pointer to search path or environment variable  */
PSZ pszFileName;     /* pointer to filename                             */
PBYTE pbBuf;         /* pointer to result buffer                        */
USHORT cbBuf;        /* length of result buffer                         */
```

The **DosSearchPath** function searches the specified search path for the given filename. The search path is a null-terminated string that consists of a sequence of directory paths separated by semicolons (;). The function searches for the filename by looking in each directory (one directory at a time) in the order given.

Parameters

fsSearch Specifies how to interpret the *pszPath* parameter and whether to search the current directory. This parameter can be a combination of the following values:

Value	Meaning
DSP_ENVIRONMENT	The *pszPath* parameter points to the name of an environment variable. The function retrieves the value of the environment variable from the environment segment of the process and uses it as the search path. If this value is not specified, *pszPath* points to a string that specifies the search path. This value cannot be used with the DSP_PATH value.
DSP_IGNORE_NET_ERR	If this value is set, the search ignores any network errors encountered during during processing and continues to search the remainder of the path. If this value is not specified, a network error (for example, when a server is unavailable) causes the search to halt.
DSP_CUR_DIRECTORY	The function searches the current directory before it searches the first directory in the search path. If this value is not specified, the function searches the current directory only if it is explicitly given in the search path.
DSP_PATH	The *pszPath* parameter points to a string that specifies the search path. This value cannot be used with the DSP_ENVIRONMENT value.

pszPath Points to the null-terminated string that specifies the search path. If DSP_PATH is specified for the *fsSearch* parameter, the *pszPath* parameter points to an environment variable. Otherwise, the *pszPath* parameter points to one or more paths to search. The paths are separated by semicolons (;).

pszFileName Points to a null-terminated string that specifies the filename to search for. The string must be a valid MS OS/2 filename and can contain wildcard characters.

pbBuf Points to the buffer that receives the full path name of the file if the filename is found.

cbBuf Specifies the length (in bytes) of the buffer pointed to by the *pbBuf* parameter.

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

Comments If **DosSearchPath** finds a matching filename in any of the directories specified by the search path, the function copies the full, null-terminated path name to the buffer pointed to by the *pbBuf* parameter. If the filename pointed to by the *pszFileName* parameter contains wildcard characters, the resulting path name will also contain wildcard characters; the **DosFindFirst** function can be used to retrieve the actual filename(s).

The **DosSearchPath** function does not check for the validity of filenames. If the filename is not valid, the function returns an error, indicating that the file was not found.

Example This example uses the search path specified by the **DPATH** environment variable to search for the *abc.txt* filename:

```
CHAR szFoundFile[128];
DosSearchPath(DSP_ENVIRONMENT,        /* uses environment variable   */
    "DPATH",                          /* uses DPATH search path      */
    "abc.txt",                        /* filename                    */
    szFoundFile,                      /* receives resulting filename */
    sizeof(szFoundFile));             /* length of result buffer     */
```

The following example is identical to the first example if the **DPATH** variable is defined as shown:

```
DPATH=c:\sysdir;c:\init
```

```
DosSearchPath(DSP_PATH,               /* uses search path            */
    "c:\\sysdir;c:\\init",            /* search path                 */
    "abc.txt",                        /* filename                    */
    szFoundFile,                      /* receives resulting filename */
    sizeof(szFoundFile));             /* length of result buffer     */
```

See Also **DosFindFirst, DosScanEnv**

Changes The constants SEARCH_PATH, SEARCH_CUR_DIRECTORY, and SEARCH_ENVIRONMENT have been changed to DSP_PATH, DSP_CUR_DIRECTORY, and DSP_ENVIRONMENT, respectively. A new constant, DSP_IGNORE_NET_ERR, has been added to allow searches to continue when a network drive specified in the path might not be available at the time of the search.

■ DosSemClear — Correction

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

The **DosSemClear** function clears a system or RAM semaphore that has been set by using the **DosSemRequest, DosSemSet,** or **DosSemSetWait** function.

Parameters

hsem Identifies the semaphore to clear. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

Return Value

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

ERROR_EXCL_SEM_ALREADY_OWNED

Comments

The **DosSemClear** function cannot clear a system semaphore that is owned by another process unless the semaphore is nonexclusive.

Example

This example uses the **DosSemClear** function to clear a RAM semaphore and a system semaphore:

```
ULONG hsem = 0;
HSYSSEM hsys;
DosSemClear(&hsem);        /* clears RAM semaphore    */
DosSemClear(hsys);         /* clears system semaphore */
```

See Also

DosCreateSem, DosMuxSemWait, DosOpenSem, DosSemRequest, DosSemSet, DosSemSetWait, DosSemWait

Corrections

The example incorrectly used the address of the system semaphore rather than the handle of the system semaphore. System semaphores require the handle of the semaphore; RAM semaphores require the address of the semaphore.

DosSemRequest — Correction

```
USHORT DosSemRequest(hsem, lTimeOut)
HSEM hsem;         /* semaphore handle */
LONG lTimeOut;     /* time-out         */
```

The **DosSemRequest** function obtains and sets a semaphore. If no previous thread has set the semaphore, **DosSemRequest** sets the semaphore and returns immediately. If the semaphore has already been set by another thread, the function waits until a thread clears the semaphore (by using the **DosSemClear** function) or until a time-out occurs. The **DosSemRequest** function is also used to obtain ownership of a system semaphore created with the CSEM_PRIVATE flag set (see **DosCreateSem**).

Parameters

hsem Identifies the semaphore to set. This value can be the handle of a system semaphore that has been previously created or opened by using the **DosCreateSem** or **DosOpenSem** function, or it can be the address of a RAM semaphore.

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

Return Value

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

ERROR_INTERRUPT
ERROR_SEM_OWNER_DIED

ERROR_SEM_TIMEOUT
ERROR_TOO_MANY_SEM_REQUESTS

Comments

If **DosSemRequest** is used to obtain exclusive ownership of a semaphore created by another process, it will wait (if *lTimeOut* is non-zero) until the semaphore is clear, or until the process that currently owns the semaphore closes the semaphore or terminates. If the process owning the semaphore terminates, **DosSemRequest** will return with an error value of ERROR_SEM_OWNER_DIED, however ownership will be transferred and the semaphore will be set and can be used by the calling process.

The effects of **DosSemRequest** are cumulative. If multiple calls to the **DosSemRequest** function set the semaphore, the same number of calls to the **DosSemClear** function are required to clear the semaphore.

If more than one thread has requested to set the semaphore, a thread may have to wait through several changes of the semaphore before it continues (depending on which thread clears the semaphore and when the system scheduler passes control to the thread). As long as the semaphore is set (even if it has been cleared and reset since the thread originally called the function), the thread must wait.

The **DosSemRequest** function 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  */
    SEM_INDEFINITE_WAIT);       /* waits indefinitely */
    .
    .
    .
DosSemClear(&hsem);             /* clears semaphore   */
```

See Also

DosCreateSem, DosExitList, DosMuxSemWait, DosOpenSem, DosSemClear, DosSemSet, DosSemSetWait, DosSemWait

Corrections

DosSemRequest is used not only to set a semaphore once it becomes clear, but also to obtain exclusive ownership of a system semaphore created with the CSEM_PRIVATE flag.

■ DosSetFileInfo — Change

```
USHORT DosSetFileInfo(hf, usInfoLevel, pInfoBuf, cbInfoBuf)
HFILE hf;              /* handle of file about which data sought */
USHORT usInfoLevel;    /* level of file data required            */
PBYTE pInfoBuf;        /* pointer to file-data buffer            */
USHORT cbInfoBuf;      /* length of file-data buffer             */
```

The **DosSetFileInfo** function sets information about a specific file. The file information consists of the date and time the file was created, the date and time it was last accessed, the date and time it was last written to, the size of the file, and its attributes. It can also be used to set extended attributes for a file.

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

Parameters

hf Identifies the file about which information is to be set. This handle must have been created by using the **DosOpen** function.

usInfoLevel Specifies the level of file information. This may be one of the following values:

Value	Meaning
FILE_INFO_1	Level-1 information request. This uses a **FILESTATUS** structure. Any date and time fields in this structure that the file system does not support should be set to zero.
FILE_INFO_2	Level-2 information request. This uses an **EAOP** structure, which contains the file's extended-attribute information.

pInfoBuf Points to the structure that contains the file information. This structure will be **FILESTATUS** or **EAOP**, depending on the *usInfoLevel* parameter.

The **FILESTATUS** structure has the following form:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

The **EAOP** structure has the following form:

```
typedef struct _EAOP {
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG  oError;
} EAOP;
```

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

cbInfoBuf Specifies the length (in bytes) of the buffer that contains the file information.

Return Value

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

ERROR_BUFFER_OVERFLOW
ERROR_DIRECT_ACCESS_HANDLE
ERROR_EA_LIST_INCONSISTENT
ERROR_INVALID_EA_NAME
ERROR_INVALID_HANDLE
ERROR_INVALID_LEVEL

Comments

DosSetFileInfo works only for files opened in a mode that allows write access.

Prior to the function being called, the **fpFEAlist** field in the **EAOP** structure should be initialized so that it points to the **FEALIST** structure that contains the relevant **FEA** structure. The **cbList** field in the **FEALIST** structure is valid, giving the size of the **FEA** structure.

A zero value in both the date and time components of a field causes that field to be unchanged. For example, if both the **fdateLastWrite** and **ftimeLastWrite** fields are zero in the **FILESTATUS** structure, both attributes of the file remain unchanged. If either of these fields are nonzero, both attributes of the file are set to the new values. If extended attributes are modified, the file's last modification date and time are changed.

See Also **DosBufReset, DosClose, DosNewSize, DosOpen, DosSetFileMode, DosQFileInfo**

Changes The constant FILE_INFO_2 has been added.

■ DosSetMaxFH Change

```
USHORT DosSetMaxFH( usHandles )
USHORT usHandles;     /* number of file handles */
```

The **DosSetMaxFH** function sets the maximum number of available file handles for the current process and any of its child processes. The number of available handles limits the number of files that can be opened at the same time.

Parameters

usHandles Specifies the maximum number of file handles provided to the calling process. The maximum value for this parameter is 32,767; the minimum is 20. This number must not be smaller than the current number of file handles allocated.

Return Value

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

ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Comments

This function preserves all currently open file handles.

There are three handles in use when a process is started—for standard input, standard output, and standard error. The number of available handles set by the **DosSetMaxFH** function includes these handles. The **DosOpenQueue, KbdOpen** and **MouOpen** functions also use these handles.

See Also **DosDupHandle, DosOpen, DosOpenQueue, KbdOpen, MouOpen**

Changes The maximum number of handles has been increased from 255 to 32,767.

■ DosSetPathInfo New

```
USHORT DosSetPathInfo( pszPathName, usInfoLevel, pInfoBuf, cbInfoBuf, fsOptions, ulReserved)
PSZ pszPathName;      /* pointer to path                    */
USHORT usInfoLevel;   /* level of information               */
PBYTE pInfoBuf;       /* pointer to buffer for information  */
USHORT cbInfoBuf;     /* length of information buffer       */
USHORT fsOptions;     /* options                            */
ULONG ulReserved;     /* must be zero                       */
```

The **DosSetPathInfo** function sets information for a specified file or directory.

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

Parameters

pszPathName Points to the null-terminated string that specifies the path of the file or directory. The string must be a valid MS OS/2 path.

usInfoLevel Specifies the level of information to set. This parameter can be one of the following values:

Value	Meaning
FIL_STANDARD	Use a **FILESTATUS** structure.
FIL_QUERYEASIZE	Use an **EAOP** structure to set extended attributes.

pInfoBuf Points to the buffer where path information is stored. The buffer contains a **FILESTATUS** structure for FIL_STANDARD information or an **EAOP** structure for FIL_QUERYEASIZE information.

The **FILESTATUS** structure has the following form:

```
typedef struct _FILESTATUS {
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
} FILESTATUS;
```

The **EAOP** structure has the following form:

```
typedef struct _EAOP {
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG    oError;
} EAOP;
```

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

cbInfoBuf Specifies the length (in bytes) of the buffer pointed to by the *pInfoBuf* parameter.

fsOptions Specifies one or more options. For MS OS/2, version 1.2, DSPI_WRTTHRU is the only available option. The DSPI_WRTTHRU option means all data, including extended attributes, must be written to the disk before the function returns.

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_BUFFER_OVERFLOW
ERROR_EA_LIST_INCONSISTENT
ERROR_FILENAME_EXCED_RANGE
ERROR_INVALID_EA_NAME
ERROR_INVALID_LEVEL
ERROR_PATH_NOT_FOUND

Comments

If the **DosSetPathInfo** function is used to set extended-attribute information, the **fpFEAList** field of the **EAOP** structure should point to the **FEALIST** structure that contains the extended attributes. The **fpGEAList** field of the **EAOP** structure will be ignored.

DosSetPathInfo fails if another process has the same file or directory.

A zero value in both the date and time fields of an attribute cause those attributes to remain unchanged. For example, if both the **fdateLastWrite** and **ftimeLastWrite** fields of the **FILESTATUS** structure are zero, both attributes are unchanged. If either field is nonzero, both fields are set to the new values. If extended attributes are modified, the file's last modification date and time will be changed.

See Also **DosQPathInfo, DosSetFileInfo**

■ DosSetPrty — Change

```
USHORT DosSetPrty( fScope, fPrtyClass, sChange, id)
USHORT fScope;        /* scope of change              */
USHORT fPrtyClass;    /* priority class to set        */
SHORT sChange;        /* change in priority level     */
USHORT id;            /* process or thread identifier */
```

The **DosSetPrty** function sets the scheduling priority of the specified process or thread by changing the priority class and/or the priority level.

Within each class, a thread's priority level may vary—either through system action or through the **DosSetPrty** function. The system changes a thread's priority level based on that thread's actions and the overall system activity.

Parameters

fScope Specifies the scope of the request. This parameter can be one of the following values:

Value	Meaning
PRTYS_PROCESS	Priority for the process and all its threads.
PRTYS_PROCESSTREE	Priority for the process and all its child processes.
PRTYS_THREAD	Priority for one thread in the current process.

fPrtyClass Specifies the priority class of a process or thread. This parameter can be one of the following values:

Value	Meaning
PRTYC_IDLETIME	Idle time.
PRTYC_NOCHANGE	No change; leave as is.
PRTYC_REGULAR	Regular.
PRTYC_FOREGROUND	Foreground server.
PRTYC_TIMECRITICAL	Time-critical.

sChange Specifies the relative change in the current priority level of the process or thread. This parameter can be any value from -31 through +31, or the constants PRTYD_MINIMUM or PRTYD_MAXIMUM, which specify the minimum and maximum change allowed.

id Specifies the process or thread identifier, depending on the value of the *fScope* parameter. If the value is a process identifier, it must be for the calling process or a child of the calling process. A value of zero can be used to specify the current thread or process.

Return Value

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

ERROR_INVALID_PCLASS
ERROR_INVALID_PDELTA
ERROR_INVALID_PROCID
ERROR_INVALID_SCOPE
ERROR_INVALID_THREADID
ERROR_NOT_DESCENDANT

Comments

The PRTYC_FOREGROUND priority is higher than PRTYC_REGULAR, but lower than PRTYC_TIMECRITICAL. PRTYC_FOREGROUND is a static priority that is not changed by the system. This allows a thread or process in a background screen group to service requests of a foreground process in a timely manner. Because the priority level is static, this priority should be used only when absolutely necessary. Indiscriminate use degrades system performance.

See Also

DosEnterCritSec, DosGetInfoSeg, DosGetPrty

Changes

A new value, PRTYC_FOREGROUND, can be specified for *fPrtyClass*.

■ DosSetVec — Correction

```
USHORT DosSetVec( usVecNum, pfnFunction, ppfnPrev)
USHORT usVecNum;     /* type of exception                              */
PFN pfnFunction;     /* pointer to function                            */
PPFN ppfnPrev;       /* pointer to variable for previous function's address */
```

The **DosSetVec** function installs or removes an exception handler for a specified exception. An exception is a program error, such as division by zero, that causes the system to pass control to the exception handler. The exception handler is an assembly-language routine that corrects errors or cleans up programs before terminating. The system calls the exception handler whenever the specified exception occurs. If a process does not install its own exception handler, the default exception handler terminates the process when an exception occurs.

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

Parameters

usVecNum Specifies the number of the exception vector. This parameter can be one of the following values:

Value	Meaning
VECTOR_DIVIDE_BY_ZERO	Division by zero
VECTOR_EXTENSION_ERROR	Processor extension error
VECTOR_INVALIDOPCODE	Invalid operation code (opcode)
VECTOR_NO_EXTENSION	Processor extension not available
VECTOR_OUTOFBOUNDS	Out of bounds
VECTOR_OVERFLOW	Overflow

pfnFunction Points to the address of the exception handler that receives control when the specified exception occurs. If this parameter is zero, the **DosSetVec** function removes the current exception handler. For a full description, see the following "Comments" section.

ppfnPrev Points to the variable that receives the address of the previous exception handler. The new exception handler can use this address to chain exception handling through all previous handlers or to restore the previous exception handler.

Return Value

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

ERROR_INVALID_FUNCTION

Comments

When the system calls the exception handler, it enables interrupts and pushes the machine status word and far return address on the stack. If the exception handler returns, it must use the **iret** (return-from-interrupt) instruction.

If the **DosSetVec** function is used to install an exception handler for the vector VECTOR_EXTENSION_ERROR, the function sets the machine status word to indicate that no 80287 processor is available. The emulation bit is set and the monitor-processor bit is cleared. (This is done without regard for the true state of the hardware.) If the **DosSetVec** function is used to remove the exception handler for VECTOR_EXTENSION_ERROR, the function sets the machine status word to reflect the true state of the hardware.

If the routine being registered is in a segment that has the **iopl** instruction indicated, the exception when it occurs, causes a general protection fault and the process is terminated.

Restrictions

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

- Because the 8086 and 8088 microprocessors do not raise this exception, *usVecNum* cannot be VECTOR_EXTENSION_ERROR.

See Also

DosDevConfig, DosError

Corrections

The exception handler must not be in an **IOPL** segment or the exception will cause a general protection fault.

■ DosShutdown New

```
USHORT DosShutdown (ulReserved)
ULONG ulReserved;     /* must be zero */
```

The **DosShutdown** function flushes all system buffers and closes down the file system. After calling **DosShutdown,** no process can access the file system until the computer is rebooted.

Parameters

ulReserved Specifies a reserved value; must be zero.

Return Value

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

ERROR_INVALID_PARAMETER

Comments

The **DosShutdown** function may take as much as several minutes to return, depending on the amount of data being written to the disk.

Because it is not possible to swap memory to the disk once the **DosShutdown** function has been called, some functions may fail due to a lack of memory in low memory situations. All memory that the calling process may need should be allocated before calling **DosShutdown**; this includes implicit memory allocation that may be done by system functions for the calling process.

■ DosStartSession Correction

```
USHORT DosStartSession(pstdata, pidSession, ppid)
PSTARTDATA pstdata;    /* pointer to structure with session data    */
PUSHORT pidSession;    /* pointer to variable for session identifier */
PUSHORT ppid;          /* pointer to variable for process identifier */
```

The **DosStartSession** function starts a session (screen group) and specifies which program to start in that session. This function creates an independent session or a child session, depending on the value of the **Related** field in the **STARTDATA** structure.

Parameters

pstdata Points to the **STARTDATA** structure that contains data describing the session to start. The **STARTDATA** structure has the following form:

```
typedef struct _STARTDATA {
    USHORT Length;
    USHORT Related;
    USHORT FgBg;
    USHORT TraceOpt;
    PSZ    PgmTitle;
    PSZ    PgmName;
    PBYTE  PgmInputs;
    PBYTE  TermQ;
    PBYTE  Environment;
    USHORT InheritOpt;
    USHORT SessionType;
    PSZ    IconFile;
    ULONG  PgmHandle;
    USHORT PgmControl;
    USHORT InitXPos;
    USHORT InitYPos;
    USHORT InitXSize;
    USHORT InitYSize;
} STARTDATA;
```

pidSession Points to the variable that receives the identifier of the child session.

ppid Points to the variable that receives the process identifier of the child process.

Return Value

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

Comments

The MS OS/2 session manager writes a data element into the specified queue when the child session created by the **DosStartSession** function terminates. A parent session can be notified when a child session has terminated by using the **DosReadQueue** function. When the child session terminates, the request value returned by **DosReadQueue** is zero, and the data-element format consists of two unsigned values: the session identifier and the result code.

Only the process that calls the **DosStartSession** function should call the **DosReadQueue** function. Only this process can address the notification data element. After reading and processing the data element, the calling process must use the **DosFreeSeg** function to free the segment that contains the data element.

A child session is created when the **Related** field of the **STARTDATA** structure is set to TRUE.

The process identifier of the child process cannot be used with MS OS/2 functions, such as **DosSetPrty**, that require a parent process/child process relationship.

An independent session is created when the **Related** field of the **STARTDATA** structure is set to FALSE. An independent session is not under the control of the starting session. The **DosStartSession** function does not copy session and process identifiers for an independent session to the *pidSession* and *ppid* parameters.

New sessions can be started in the foreground only when the caller's session (or one of the caller's descendant sessions) is currently executing in the foreground. The new session appears in the shell switch list.

See Also

DosCreateQueue, DosExecPgm, DosFreeSeg, DosReadQueue, DosSelectSession, DosSetPrty, DosSetSession, DosStopSession

Corrections

The comments incorrectly stated that an independent session was created when the **Related** field of the **STARTDATA** structure is set to TRUE. The **Related** field must be set to FALSE to create an independent session.

■ DosSubAlloc — Correction

```
USHORT DosSubAlloc( sel, pusOffset, cbBlock)
SEL sel;                  /* segment selector                  */
PUSHORT pusOffset;        /* pointer to variable for offset    */
USHORT cbBlock;           /* requested size of memory block    */
```

The **DosSubAlloc** function allocates memory in a segment that was allocated previously by using the **DosAllocSeg** or **DosAllocShrSeg** function and that was initialized by using the **DosSubSet** function.

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

Parameters

sel Specifies the selector of the data segment in which the memory should be allocated.

pusOffset Points to the variable that receives the offset to the allocated block.

cbBlock Specifies the size (in bytes) of the requested memory block.

Return Value

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

ERROR_DOSSUB_BADSIZE
ERROR_DOSSUB_NOMEM

Comments

The *cbBlock* parameter must not be greater than the maximum size of the segment minus 8 bytes. Since all memory blocks are aligned on byte boundaries, the *cbBlock* parameter does not need to be a multiple of 16; however, it will be rounded to a multiple of 4.

DosSubAlloc can be issued from ring 2 or ring 3; the suballocation segment can be either a ring-2 or a ring-3 segment.

See Also **DosAllocSeg, DosAllocShrSeg, DosSubFree, DosSubSet**

Corrections The *cbBlock* parameter is rounded to a multiple of 4 before being processed.

■ DosSubFree Correction

```
USHORT DosSubFree( sel, offBlock, cbBlock)
SEL sel;              /* segment selector                */
USHORT offBlock;      /* block offset                    */
USHORT cbBlock;       /* number of bytes in block to free */
```

The **DosSubFree** function frees memory that was allocated previously by using the **DosSubAlloc** function.

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

Parameters

sel Specifies the selector of the data segment from which the memory should be freed.

offBlock Specifies the offset of the memory block to be freed. This offset must have been created previously by using the **DosSubAlloc** function.

cbBlock Specifies the size (in bytes) of the block to free. This parameter should by a multiple of 4. If it is not, it will be rounded prior to being used by this function.

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

ERROR_DOSSUB_BADSIZE
ERROR_DOSSUB_OVERLAP

Comments **DosSubFree** can be issued from ring 2 or ring 3; and the suballocation segment can be either a ring-2 or a ring-3 segment.

See Also **DosAllocSeg, DosSubAlloc, DosSubSet**

Corrections The *cbBlock* parameter is rounded to a multiple of 4 before being processed.

■ DosWaitNmPipe Correction

```
USHORT DosWaitNmPipe( pszName, ulTimeOut)
PSZ pszName;          /* pointer to pipe name */
ULONG ulTimeOut;      /* time-out value       */
```

The **DosWaitNmPipe** function waits for a named pipe to become available.

Parameters

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

ulTimeOut Specifies the amount of time (in milliseconds) MS OS/2 should wait for the pipe to become available. A value of NP_INDEFINITE_WAIT causes an infinite wait. A value of NP_DEFAULT_WAIT causes the system to wait for the default time specified by the call to the **DosMakeNmPipe** function call that created this named pipe.

Return Value

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

ERROR_BAD_PIPE
ERROR_INTERRUPT
ERROR_SEM_TIMEOUT

Comments

The **DosWaitNmPipe** function should be used only when the **DosOpen** function returns the ERROR_PIPE_BUSY error value.

If more than one process has requested a named pipe that has become available, MS OS/2 gives the pipe to the process that has been waiting the longest.

See Also

DosOpen

Corrections

A value of NP_INDEFINITE_WAIT for the *ulTimeOut* parameter specifies an infinite wait; a value of NP_DEFAULT_WAIT for *ulTimeOut* uses the default time-out specified in the **DosMakeNmPipe** function.

■ DosWrite — Correction

```
USHORT DosWrite( hf, pvBuf, cbBuf, pcbBytesWritten )
HFILE hf;                      /* file handle                               */
PVOID pvBuf;                   /* pointer to buffer                         */
USHORT cbBuf;                  /* number of bytes to write                  */
PUSHORT pcbBytesWritten;       /* pointer to variable receiving byte count */
```

The **DosWrite** function writes data from a buffer to a file, then copies the number of bytes written to a variable.

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

Parameters

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

pvBuf Points to the buffer that contains the data to write.

cbBuf Specifies the number of bytes to write.

pcbBytesWritten Points to the variable receiving the number of bytes written.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NOT_DOS_DISK
ERROR_WRITE_FAULT

Comments

The **DosWrite** function begins to write at the current file-pointer position. The file-pointer position can be changed by using the **DosChgFilePtr** function.

If the specified file has been opened using the write-through flag, the **DosWrite** function writes data to the disk before returning. Otherwise, the system collects the data in an internal file buffer and writes the data to the disk only when the buffer is full.

The **DosWrite** function may write fewer bytes to the file than the number specified in the *cbBuf* parameter if there is not enough space on the disk for all of the requested bytes. The *cbBuf* parameter can be zero without causing an error—that is, writing no bytes is acceptable.

The efficiency with which **DosWrite** writes to a disk is improved when *cbBuf* is set to a multiple of the disk's bytes-per-sector size. When *cbBuf* is set this way, **DosWrite** writes directly to the disk, without first copying the data to an internal file buffer. (**DosQFSInfo** retrieves the bytes-per-sector value for a disk.)

DosWrite can be used to write bytes or messages to a pipe. Each write to a message pipe writes a message whose size is the length of the write; **DosWrite** automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

Writes in blocking mode always write all requested bytes before returning. In nonblocking mode, writes return either with all bytes written or none written; the latter will occur in cases where **DosWrite** would have to block in order to complete the request—for example, if there is no room in the pipe buffer or if the buffer is currently being written to by another client).

An attempt to write to a pipe whose other end has been closed will return the error ERROR_BROKEN_PIPE.

Example

This example creates the file *abc* and calls the **DosWrite** function to write the contents of the *abBuf* buffer to the file:

```
BYTE abBuf[512];
HFILE hf;
USHORT usAction, cbBytesWritten, usError;
usError = DosOpen("abc", &hf, &usAction, OL, FILE_NORMAL,
    FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, OL);
if (!usError) {
    DosWrite(hf,                      /* file handle            */
        abBuf,                        /* buffer address         */
        sizeof(abBuf),                /* buffer size            */
        &cbBytesWritten);             /* address of bytes written */
```

See Also

DosChgFilePtr, DosOpen, DosQFSInfo, DosRead, DosWriteAsync

Corrections

DosWrite can be used to write bytes or messages to a pipe. Relevant information about writing to a named pipe has been added.

■ DosWriteAsync — Correction

```
USHORT DosWriteAsync(hf, hsemRam, pusErrCode, pvBuf, cbBuf, pcbBytesWritten)
HFILE hf;                  /* file handle                                */
PULONG hsemRam;            /* pointer to RAM semaphore                   */
PUSHORT pusErrCode;        /* pointer to variable for error value        */
PVOID pvBuf;               /* pointer to buffer containing data to write */
USHORT cbBuf;              /* number of bytes in buffer                  */
PUSHORT pcbBytesWritten;   /* pointer to variable for bytes written      */
```

The **DosWriteAsync** function writes one or more bytes of data to a specified file. The function writes the data asynchronously—that is, the function returns immediately, but continues to copy data to the specified file while the process continues with other tasks.

Parameters

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

hsemRam Points to the RAM semaphore that indicates when the function has finished reading the data.

pusErrCode Points to the variable that receives an error value.

pvBuf Points to the buffer that contains the data to write.

cbBuf Specifies the number of bytes to write.

pcbBytesWritten Points to the variable receiving the number of bytes written.

Return Value

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

ERROR_ACCESS_DENIED
ERROR_BROKEN_PIPE
ERROR_INVALID_HANDLE
ERROR_LOCK_VIOLATION
ERROR_NO_PROC_SLOTS
ERROR_NOT_DOS_DISK
ERROR_WRITE_FAULT

Comments

The **DosWriteAsync** function starts writing at the current file-pointer position. The file-pointer position can be changed by using the **DosChgFilePtr** function.

If the specified file has been opened using the write-through flag, the **DosWriteAsync** function writes data to an internal file buffer and to the disk before returning. If the write-through flag has not been set, the system collects the data in an internal file buffer and writes the data to the disk only when the buffer is full.

The **DosWriteAsync** function may write fewer bytes to the file than the number specified in the *cbBuf* parameter if there is not enough space on the disk for all the requested bytes. The *cbBuf* parameter can be zero without causing an error—that is, writing no bytes is acceptable.

When the **DosWriteAsync** function has written the data, it clears the RAM semaphore pointed to by the *hsemRam* parameter. If the process uses the semaphore to determine when data is available, it must use the **DosSemSet** function to set the semaphore before calling **DosWriteAsync.**

The efficiency with which the **DosWriteAsync** function writes to a disk is improved when the *cbBuf* parameter is set to a multiple of the disk's bytes-per-sector size. When *cbBuf* is set this way, the function writes directly to the disk, without first copying the data to an internal file buffer. (The **DosQFSInfo** function retrieves the byters-per-sector value for a disk.)

DosWriteAsync can be used to write bytes or messages to a pipe. Each write to a message pipe writes a message whose size is the length of the write; **DosWriteAsync** automatically encodes message lengths in the pipe, so applications need not encode this information in the buffer being written.

In blocking mode, write operations always write all requested bytes before returning. In nonblocking mode, write operations return either with all bytes written or none written; the latter occurs in cases where **DosWriteAsync** has to block in order to complete the request (for example, if there is no room in the pipe buffer or if the buffer is currently being written to by another process).

When the function tries to write to a pipe whose other end has been closed, it returns the error ERROR_BROKEN_PIPE.

Example

This example creates the file *abc.ext*, sets a RAM semaphore, and calls the **DosWriteAsync** function to write the contents of the buffer *abBuf* to a file. When any additional processing is finished, the example calls the **DosSemWait** function to wait until **DosWriteAsync** has finished writing to the file.

```
ULONG hsemWrite = 0;
BYTE abBuf[1024];
HFILE hf;
USHORT usAction, cbBytesWritten;
USHORT usWriteAsyncError;
DosOpen("abc.ext", &hf, &usAction, 0L, FILE_NORMAL,
    FILE_CREATE,
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, 0L);

DosSemSet(&hsemWrite);          /* sets semaphore           */
DosWriteAsync(hf,               /* file handle              */
    &hsemWrite,                 /* semaphore address        */
    &usWriteAsyncError,         /* return-code address      */
    abBuf,                      /* buffer address           */
    sizeof(abBuf),              /* buffer size              */
    &cbBytesWritten);           /* address of bytes written */
    .
    . /* Other processing would go here. */
    .
DosSemWait(&hsemWrite, -1L);    /* waits for DosWriteAsync  */
if (usWriteAsyncError) {
    .
    . /* Error processing would go here. */
    .
```

See Also

DosChgFilePtr, DosOpen, DosQFSInfo, DosReadAsync, DosSemSet, DosSemWait, DosWrite

Corrections

Information about using **DosWriteAsync** with named pipes has been added.

■ DosWriteQueue — Correction

```
USHORT DosWriteQueue(hqueue, usRequest, cbBuf, pbBuf, usPriority)
HQUEUE hqueue;       /* target-queue handle                   */
USHORT usRequest;    /* request/identification data           */
USHORT cbBuf;        /* number of bytes to write              */
PBYTE pbBuf;         /* pointer to buffer with element to write */
UCHAR usPriority;    /* priority of element to write          */
```

The **DosWriteQueue** function writes an element to the specified queue. The position of the element in the queue is determined by the value specified in the *fQueueOrder* parameter of the **DosCreateQueue** function when the queue was created; if this parameter was set to 0x0002 (priority queue), the *usPriority* parameter of the **DosWriteQueue** function can be used to set the priority of the element. After the element is written, the process that owns the queue can read the element by using the **DosPeekQueue** or **DosReadQueue** function.

Parameters

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

usRequest Specifies a program-supplied event code. MS OS/2 does not use this field; it is reserved for the program's use. The queue owner can retrieve this value by using the **DosPeekQueue** or **DosReadQueue** function.

cbBuf Specifies the number of bytes to be copied from the buffer pointed to by the *pbBuf* parameter.

pbBuf Points to the buffer that contains the element to be written to the queue.

usPriority Specifies the element priority. This parameter can be any value from 0 through 15; 15 is the highest priority.

Return Value

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

ERROR_QUE_INVALID_HANDLE
ERROR_QUE_NO_MEMORY

Comments

The **DosWriteQueue** function returns an error value if the queue has been closed by the process that owns it.

If the queue owner uses a RAM semaphore to notify it when elements are added to the queue, the semaphore must be shared. If the notifying semaphore is a system semaphore, the writing process must have opened the semaphore by using the **DosOpenSem** function.

Example

This example opens a queue called \queues\queuename. In order to write to the queue, the process allocates shared memory, gives the memory to the queue owner, copies data to the shared memory, and calls **DosWriteQueue**. The process then frees the shared memory. The queue owner must also free the shared memory before it becomes available to the system again. For more information, see **DosReadQueue**.

```
PID pidOwner;
SEL sel, selRecipient;

DosOpenQueue(&pidOwner, &hqueue,
    "\\queues\\queuename");                 /* opens queue            */
DosAllocSeg(512, &sel, SEG_GIVEABLE);       /* allocates shared memory */
DosGiveSeg(sel, pidOwner, &selRecipient);   /* gives it to queue owner */
    .
    . /* Copy the data to the shared memory segment. */
    .
       DosWriteQueue(hqueue,      /* queue handle             */
           0,                     /* request data             */
           11,                    /* length of data           */
    MAKEP(selRecipient, 0),       /* data buffer              */
           0);                    /* element priority         */
       DosFreeSeg(sel);           /* frees shared memory segment */
```

See Also

DosCreateQueue, DosOpenQueue, DosOpenSem, DosPeekQueue, DosReadQueue

Corrections

The example worked only for interthread communication. It has been replaced with an example that works for interprocess communication.

The description of the *cbBuf* parameter incorrectly stated that this parameter contained the number of bytes to be written to the buffer. It now correctly states that this is the number of bytes to be written from the buffer.

■ EM_QUERYREADONLY New

```
EM_QUERYREADONLY
mp1 = OL;     /* not used, must be zero */
mp2 = OL;     /* not used, must be zero */
```

An application sends the EM_QUERYREADONLY message to retrieve the read-only state of an entry field.

Parameters This message does not use any parameters.

Return Value The return value is TRUE if the read-only state is set; otherwise it is FALSE.

See Also EM_SETREADONLY

■ EM_SETINSERTMODE New

```
EM_SETINSERTMODE
mp1 = MPFROMSHORT(fInsertMode);     /* insert-mode flag       */
mp2 = OL;                           /* not used, must be zero */
```

An application sends the EM_SETINSERTMODE message to set or clear the system insert-mode state.

Parameters *fInsertMode* Low word of *mp1*. Specifies whether to set or clear the insert mode. If this parameter is TRUE, insert mode is turned on; if it is FALSE, insert mode is turned off.

Return Value The return value is TRUE if the previous insert mode was on or FALSE if the previous insert mode was off.

Comments This message changes the SV_INSERTMODE system constant to reflect the current insert-mode state. It also sends an EN_INSERTMODETOGGLE notification message.

See Also EN_INSERTMODETOGGLE

■ EM_SETREADONLY New

```
EM_SETREADONLY
mp1 = MPFROMSHORT(fReadOnly);     /* read-only state        */
mp2 = OL;                         /* not used, must be zero */
```

An application sends the EM_SETREADONLY message to set the read-only state of an entry field.

Parameters *fReadOnly* Low word of *mp1*. Specifies whether to set or remove the read-only state of the entry field. A value of TRUE sets the state.

Return Value The return value is TRUE if the read-only state is set; otherwise, it is FALSE.

Comments When the read-only state of an entry field is set, the user cannot change the text within the entry field.

See Also EM_QUERYREADONLY

■ EM_SETTEXTLIMIT — Change

```
EM_SETTEXTLIMIT
mp1 = MPFROMSHORT((SHORT) cchMax);      /* max. number of bytes   */
mp2 = 0L;                               /* not used, must be zero */
```

An application sends an EM_SETTEXTLIMIT message to set the maximum number of bytes an entry-field control can contain.

Parameters *cchMax* Low word of *mp1*. Specifies the maximum number of bytes an entry field can hold.

Return Value The return value is TRUE if the operation is successful or FALSE if there is not enough memory to hold the requested number of characters.

Comments Sending an EM_SETTEXTLIMIT message causes memory to be allocated from the control heap for the specified maximum number of bytes. Failure to allocate sufficient memory results in a WM_CONTROL message, with the EN_MEMERROR notification code being sent to the owner window.

See Also WM_CONTROL

Changes All references to characters have been replaced by bytes to accommodate systems in which a character may be composed of more than one byte.

■ EN_CHANGE — New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID          */
usNotifyCode = EN_CHANGE;
hwndEdit = HWNDFROMMP(mp2);          /* window handle of entry field */
```

The EN_CHANGE notification message is sent when the text in an entry field changes.

Parameters *id* Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_CHANGE.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

Return Value An application should return zero if it processes this message.

See Also WM_CONTROL

■ EN_INSERTMODETOGGLE — New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID          */
usNotifyCode = EN_INSERTMODETOGGLE
hwndEdit = HWNDFROMMP(mp2);          /* window handle of entry field */
```

The EN_INSERTMODETOGGLE notification message is sent when the insert mode of an entry-field control is toggled.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_INSERTMODETOGGLE.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

Return Value An application should return zero if it processes this message.

See Also EM_SETINSERTMODE, WM_CONTROL

■ EN_KILLFOCUS New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);      /* control-window ID         */
usNotifyCode = EN_KILLFOCUS;
hwndEdit = HWNDFROMMP(mp2);           /* window handle of entry field */
```

The EN_KILLFOCUS notification message is sent when an entry-field control loses the input focus.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_KILLFOCUS.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

See Also EN_SETFOCUS, WM_CONTROL

■ EN_MEMERROR New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);      /* control-window ID         */
usNotifyCode = EN_MEMERROR;
hwndEdit = HWNDFROMMP(mp2);           /* window handle of entry field */
```

The EN_MEMERROR notification message is sent when an entry-field control cannot allocate the memory necessary to accommodate window text of the length specified by the EM_SETTEXTLIMIT message.

Parameters

id Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_MEMERROR.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

See Also EM_SETTEXTLIMIT, WM_CONTROL

■ EN_OVERFLOW New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);      /* control-window ID         */
usNotifyCode = EN_OVERFLOW
hwndEdit = HWNDFROMMP(mp2);           /* window handle of entry field */
```

The EN_OVERFLOW notification message is sent when the text limit in an entry field is exceeded.

Parameters *id* Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_OVERFLOW.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

Return Value An application should return TRUE to retry the operation.

See Also WM_CONTROL

■ EN_SCROLL New

```
EN_SCROLL
id = (USHORT) SHORT1FROMMP(mp1);      /* control-window ID          */
usNotifyCode = EN_SCROLL;
hwndEdit = HWNDFROMMP(mp2);           /* window handle of entry field */
```

The EN_SCROLL notification message is sent to the owner of the entry-field window when a scroll-bar event occurs.

Parameters *id* Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_SCROLL.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

Return Value An application should return zero if it processes this message.

See Also WM_CONTROL

■ EN_SETFOCUS New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);     /* control-window ID          */
usNotifyCode = EN_SETFOCUS;
hwndEdit = HWNDFROMMP(mp2);          /* window handle of entry field */
```

The EN_SETFOCUS notification message notifies an application when an entry field receives the input focus.

Parameters *id* Low word of *mp1*. Identifies the control window.

usNotifyCode High word of *mp1*. Set to EN_SETFOCUS.

hwndEdit Low and high word of *mp2*. Identifies the entry-field window.

See Also EN_KILLFOCUS, WM_CONTROL

■ GpiCallSegmentMatrix — Correction

```
LONG GpiCallSegmentMatrix(hps, idSegment, cElements, pmatlf, lType)
HPS hps;               /* presentation-space handle              */
LONG idSegment;        /* segment identifier                     */
LONG cElements;        /* number of matrix elements to examine   */
PMATRIXLF pmatlf;      /* address of structure for matrix        */
LONG lType;            /* transformation modifier                */
```

The **GpiCallSegmentMatrix** function draws the specified segment using an instance transformation. The function combines the instance transformation pointed to by *pmatlf* with the current model transformation, then draws the segment as if calling the **GpiDrawSegment** function. The combined transformation applies only while the function draws the segment. **GpiCallSegmentMatrix** does not modify the current model transformation.

Parameters

hps Identifies the presentation space.

idSegment Specifies the segment to draw. This value must be greater than zero.

cElements Specifies the number of matrix elements pointed to by *pmatlf*. It can be any value from 0 through 9.

pmatlf Points to a **MATRIXLF** structure that contains the matrix for the instance transformation. Although a transformation requires nine matrix elements, the function copies from the structure only the number of matrix elements specified by *cElements*. If *cElements* is less than nine, the function supplies the remaining elements by substituting corresponding elements from the identity matrix.

The **MATRIXLF** structure has the following form:

```
typedef struct _MATRIXLF {
    FIXED fxM11;
    FIXED fxM12;
    LONG  lM13;
    FIXED fxM21;
    FIXED fxM22;
    LONG  lM23;
    LONG  lM31;
    LONG  lM32;
    LONG  lM33;
} MATRIXLF;
```

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

lType Specifies how to combine the instance transformation with the model transformation. It can be one of the following values:

Value	Meaning
TRANSFORM_ADD	Adds the model transformation to the instance transformation (MODEL * INSTANCE).
TRANSFORM_PREEMPT	Adds the instance transformation to the model transformation (INSTANCE * MODEL).
TRANSFORM_REPLACE	Replaces the model transform with the instance transformation.

Return Value

The return value is GPI_OK or GPI_HITS if the function is successful (it is GPI_HITS if the detectable attribute is set for the presentation space and a correlation hit occurs). The return value is GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_CALLED_SEG_IS_CURRENT
PMERR_CALLED_SEG_NOT_FOUND
PMERR_INV_HPS
PMERR_INV_LENGTH_OR_COUNT
PMERR_INV_MATRIX_ELEMENT
PMERR_INV_MICROPS_FUNCTION
PMERR_INV_SEG_NAME
PMERR_INV_TRANSFORM_TYPE
PMERR_PS_BUSY
PMERR_SEG_CALL_RECURSIVE
PMERR_SEG_NOT_FOUND

Example

This example calls the **GpiCallSegmentMatrix** function to draw a segment three times. Each time the segment is drawn, the instance transformation doubles in size. The result is three triangles with the last triangle twice the size of the second, and the second twice the size of the first.

```
POINTL ptlStart = { 0, 0 };
POINTL ptlTriangle[] = { 100, 100, 200, 0, 0, 0 };
MATRIXLF matlfInstance = { MAKEFIXED(1, 0),  MAKEFIXED(0, 0), 0,
                           MAKEFIXED(0, 0),  MAKEFIXED(1, 0), 0,
                           0,                0,               1 };

GpiOpenSegment(hps, 1L);                 /* opens segment               */
GpiMove(hps, &ptlStart);                 /* moves to start point (0, 0) */
GpiPolyLine(hps, 3L, ptlTriangle);       /* draws triangle              */
GpiCloseSegment(hps);                    /* closes segment              */

for (i = 0; i < 3; i++) {

    /*
     * Draw the segment after adding the matrix to the model
     * transformation.
     */

    GpiCallSegmentMatrix(hps, 1L, 9, &matlfInstance, TRANSFORM_ADD);
    matlfInstance.fxM11 *= 2;
    matlfInstance.fxM22 *= 2;
}
```

See Also

GpiDrawSegment

Corrections

In the example, the **MAKEFIXED** macro is required to create **FIXED** values for initializing the structure.

■ GpiCreateLogFont — Correction

```
LONG GpiCreateLogFont( hps, pchName, lcid, pfat)
HPS hps;              /* presentation-space handle               */
PSTR8 pchName;        /* address of logical-font name            */
LONG lcid;            /* local identifier                        */
PFATTRS pfat;         /* address of structure for font attributes */
```

The **GpiCreateLogFont** function creates a logical font. A logical font is a list of font attributes, such as face name, average width, and maximum height, that an application uses to request a physical font. A physical font is the bitmap or vector information the system uses to draw characters on a device. Applications create logical fonts to specify the fonts they need, and the system maps the logical fonts to matching physical fonts.

GpiCreateLogFont creates a logical font using the font attributes specified in the structure pointed to by the *pfat* parameter. Each logical font has a local identifier and logical font name, specified by the *lcid* and *pchName* parameters, to uniquely identify it. The local identifier can then be used in subsequent graphics functions to identify the font.

Since a physical font that exactly matches the logical font may not be available, the system usually maps the logical font to the closest matching physical font. The system uses rules to map the font—for example, it chooses a font with a greater height if a font of the exact height is not available. An application can force the system to choose a particular font by setting the value of the **lMatch** field in the **FATTRS** structure to be that returned for the desired font by the **GpiQueryFonts** function. After **GpiCreateLogFont** chooses the physical font, this choice does not change for a particular logical font.

Parameters

hps Identifies the presentation space.

pchName Points to an 8-character logical-font name. It can be NULL, if no logical font name is desired.

lcid Specifies the local identifier that the application uses to refer to this font. It must be in the range 1 through 254. It is an error if this parameter is already in use to refer to a font or bitmap.

pfat Points to a **FATTRS** structure that will contain the attributes of the logical font that is created. The **FATTRS** structure has the following form:

```
typedef struct _FATTRS {
    USHORT  usRecordLength;
    USHORT  fsSelection;
    LONG    lMatch;
    CHAR    szFaceName[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lMaxBaselineExt;
    LONG    lAveCharWidth;
    USHORT  fsType;
    SHORT   sQuality;
    USHORT  fsFontUse;
} FATTRS;
```

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

Return Value

The return value is FONT_MATCH if a matching font is found, FONT_DEFAULT if a matching font could not be found, or zero if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_FONT_NOT_LOADED
PMERR_INV_FONT_ATTRS
PMERR_INV_HPS
PMERR_INV_SETID
PMERR_KERNING_NOT_SUPPORTED
PMERR_PS_BUSY
PMERR_SETID_IN_USE

Comments

To choose the system default font, set the face name to NULL and all other attributes in the **FATTRS** structure, except the code page, to zero.

To use a font, the application sets the font for the presentation space by specifying the local identifier for the corresponding logical font with the **GpiSetCharSet** function. Once a font is set, the system uses the font for subsequent text output.

Example

This example uses the **GpiCreateLogFont** function to create a logical font with the local identifier 1. The logical font has the face name "Courier" and requested width and height of 12 pels. Once the font is created, the example sets the font using the local identifier and displays a string in the font at the point (100,100).

```
USHORT i;
POINTL ptl = { 100, 100 };
FATTRS fat;

fat.usRecordLength = sizeof(FATTRS); /* sets size of structure      */
fat.fsSelection = 0;                 /* uses default selection      */
fat.lMatch = 0L;                     /* does not force match        */
fat.idRegistry = 0;                  /* uses default registry       */
fat.usCodePage = 850;                /* code-page 850               */
fat.lMaxBaselineExt = 12L;           /* requested font height is 12 pels */
fat.lAveCharWidth = 12L;             /* requested font width is 12 pels  */
fat.fsType = 0;                      /* uses default type           */
fat.fsFontUse = FATTR_FONTUSE_NOMIX; /* does not mix with graphics  */

/* Copy Courier to szFacename field. */

for (i=0; fat.szFacename[i] = "Courier"[i]; i++);


GpiCreateLogFont(hps,                /* presentation space          */
                 NULL,               /* does not use logical font name */
                 1L,                 /* local identifier            */
                 &fat);              /* structure with font attributes */

GpiSetCharSet(hps, 1L);              /* sets font for presentation space */
GpiCharStringAt(hps, &ptl, 5L, "Hello"); /* displays a string       */
```

See Also

GpiCharStringAt, GpiCreateLogFont, GpiQueryFonts, GpiSetCharSet

Corrections

In the example, the fat.fsType field should be set to 0 rather than to FATTR_TYPE_FIXED.

GpiDestroyPS — Correction

```
BOOL GpiDestroyPS(hps)
HPS hps;     /* presentation-space handle */
```

The **GpiDestroyPS** function destroys the presentation space and releases all resources owned by the presentation space. This function should only be used to destroy presentation spaces created by the **GpiCreatePS** function.

Parameters

hps Identifies the presentation space.

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HPS
PMERR_PS_BUSY
PMERR_PS_IS_ASSOCIATED

Example

This example uses the **GpiDestroyPS** function to destroy the presentation space associated with a memory device context:

```
HDC hdc;
HPS hps;
SIZEL page = { 0, 0 };

/* Create the memory device context and presentation space. */

hdc = DevOpenDC(hab, OD_MEMORY, "*", 0L, NULL, NULL);
hps = GpiCreatePS(hab, hdc, &page, PU_PELS | GPIT_MICRO | GPIA_ASSOC);
    .
    .
    .
GpiAssociate(hps, NULL);
GpiDestroyPS(hps);     /* destroys presentation space */
DevCloseDC(hdc);       /* closes device context       */
```

See Also

GpiCreatePS

Corrections

In the example, **GpiAssociate** must be called before **DevCloseDC**. This is true whenever a device context is associated with a presentation space.

GpiGetData — Correction

```
LONG GpiGetData(hps, idSegment, poff, cmdFormat, cb, pb)
HPS hps;            /* presentation-space handle              */
LONG idSegment;     /* segment identifier                     */
PLONG poff;         /* address of variable for segment offset */
LONG cmdFormat;     /* conversion type                        */
LONG cb;            /* length in bytes of the data buffer     */
PBYTE pb;           /* address of buffer for data             */
```

The **GpiGetData** function copies graphics orders from the specified segment to the specified buffer. The function continues to copy the graphics orders from the segment to the buffer until all orders in the segment have been copied or the number of bytes specified by the *cb* parameter have been copied. If the function fills the buffer, the last order in the buffer may not be complete since the function does not stop on an order boundary when copying to the buffer. In any case, the function returns the number of bytes copied to the buffer.

The function starts copying graphics-order data from the location specified by the *poff* parameter. If this parameter is zero, the function copies from the beginning of the segment. After copying the data, the function replaces the value in *poff* with the offset to the next byte of data to copy from the segment (if any). This value can be used to specify the next location to copy.

The **GpiGetData** function cannot be used to copy data from an open segment, but it can be used to copy data while some other segment is open.

Parameters

hps Identifies the presentation space.

idSegment Specifies the segment identifier.

poff Points to the variable that contains the offset from the beginning of the segment to the next byte of graphics order data to copy. If this parameter is zero, the function copies from the beginning of the segment.

cmdFormat Specifies the coordinate conversion type. It can be one of the following values:

Value	Meaning
DFORM_NOCONV	Copies coordinates without converting. The coordinates are in the format used by the presentation space.
DFORM_PCLONG	Converts coordinates to PC-format long (4-byte) integers.
DFORM_PCSHORT	Converts coordinates to PC-format short (2-byte) integers.
DFORM_S370SHORT	Converts coordinates to S/370-format short (2-byte) integers.

cb Specifies the length in bytes of the buffer to receive the graphics orders.

pb Points to the buffer that receives the graphics-order data.

Return Value

The return value is the number of graphics-order bytes copied if the function is successful or GPI_ALTERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_DATA_TOO_LONG
PMERR_INV_GETDATA_CONTROL
PMERR_INV_HPS
PMERR_INV_LENGTH
PMERR_INV_LENGTH_OR_COUNT
PMERR_INV_MICROPS_FUNCTION
PMERR_INV_SEG_NAME
PMERR_INV_SEG_OFFSET
PMERR_PS_BUSY
PMERR_SEG_IS_CURRENT
PMERR_SEG_NOT_FOUND

Example

This example uses the **GpiGetData** function to copy data from one segment to another:

```
LONG fFormat = DFORM_NOCONV;     /* does not convert coordinates      */
LONG offSegment = OL;            /* offset in segment                 */
LONG offNextElement = OL;        /* offset in segment to next element */
LONG cb = OL;                    /* bytes retrieved                   */
BYTE abBuffer[512];

GpiOpenSegment(hps, 3L);         /* opens segment to receive data     */
do {
    offSegment += cb;
    offNextElement = offSegment;
    cb = GpiGetData(hps, 2L, &offNextElement, fFormat, 512L, abBuffer);

    /* put data in other segment */

    if (cb > OL) GpiPutData(hps,   /* presentation-space handle       */
        fFormat,                   /* format of coordinates           */
        &cb,                       /* number of bytes in buffer       */
        abBuffer);                 /* buffer with graphics-order data */

} while (cb > 0);
GpiCloseSegment(hps);            /* closes segment that received data */
```

See Also

GpiPutData

Corrections

The *poff* parameter is a pointer to the variable that contains the offset; the *cmdFormat* parameter is an integer that specifies the conversion format.

■ GpiLoadFonts — Correction

BOOL GpiLoadFonts(*hab, pszFileName* **)**
HAB *hab*; /* anchor-block handle */
PSZ *pszFileName*; /* pointer to filename */

The **GpiLoadFonts** function loads fonts from the specified resource file. Once loaded, the fonts are private fonts and can be used by any thread in the process. Any other process can use the fonts but only if it also loads the font by using the **GpiLoadFonts**. The function loads a copy of the fonts once only. Any subsequent call to the function by another process for the same fonts simply increments the use count for the resource and gives that process access.

Parameters

hab Identifies the anchor block.

pszFileName Points to a null-terminated string. This string must be a valid MS OS/2 filename. If it does not specify a path and the filename extension, the function appends the default extension (*.dll*) and searches for the font resource file in the directories specified by the **libpath** command in the *config.sys* file.

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurred.

Error

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_INV_FONT_FILE_DATA

Example This example uses the **GpiLoadFonts** function to load all fonts from the font resource file *helv.dll*. The **GpiQueryFonts** function retrieves the number of fonts loaded.

```
LONG cFonts = OL;

GpiLoadFonts(hab, "helv");
cFonts = GpiQueryFonts(hps, QF_PRIVATE, NULL, &cFonts, OL, NULL);
```

See Also **GpiCreateLogFont, GpiDeleteSetId, GpiQueryFonts, GpiUnloadFonts**

Corrections In the example, the function loads fonts from the *helv.dll* file, not the *helv.fon* file. If no path and filename extension are given, the function by default searches for a file that has the *.dll* extension.

■ GpiOutlinePath New

```
LONG GpiOutlinePath( hps, lPath, lOptions)
HPS hps;          /* presentation-space handle   */
LONG lPath;       /* identifies path to be outlined */
LONG lOptions;    /* reserved, must be zero      */
```

The **GpiOutlinePath** function draws an outline of a path using the current line attributes. This function draws the outline such that each line, curve, and other item in the path appears to be drawn individually; it does not close the path. **GpiOutlinePath** draws the path using the current cosmetic line width (see the **GpiSetLineWidth** function); it does not fill the path. **GpiOutlinePath** deletes the path after drawing the outline.

Parameters

hps Identifies the presentation space.

lPath Identifies the path to be outlined. For MS OS/2, version 1.2, this parameter must be set to 1.

lOptions Specifies outline options. For MS OS/2, version 1.2, this parameter must be set to zero.

Return Value The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HPS
PMERR_INV_PATH_ID
PMERR_INV_RESERVED_FIELD
PMERR_PATH_UNKNOWN
PMERR_PS_BUSY

Comments If character strings are in the path, the function draws the outline of each character but does not fill the interior of the character, giving the appearance of hollow characters. For small characters, outlining in this way can give a visual appearance similar to filled characters, but with improved performance.

See Also **GpiBeginPath, GpiEndPath, GpiSetLineWidth**

■ GpiPlayMetaFile — Correction

```
LONG GpiPlayMetaFile( hps, hmf, cOptions, alOptions, pcSegments, cchDesc, pszDesc )
HPS hps;                /* presentation-space handle              */
HMF hmf;                /* metafile handle                        */
LONG cOptions;          /* number of elements in array            */
PLONG alOptions;        /* address of array of load options       */
PLONG pcSegments;       /* address of count of renumbered segments */
LONG cchDesc;           /* number of bytes in record              */
PSZ pszDesc;            /* address of buffer for descriptive record */
```

The **GpiPlayMetaFile** function plays the metafile specified by the *hmf* parameter. The function plays the metafile file by converting the graphics data in the file to graphics operations for the given presentation space. The function uses the load options specified by the *alOptions* parameter to determine how to prepare the presentation space for playing the metafile. This may include resetting the presentation space, replacing tagged bitmaps and logical fonts, and replacing the logical color table.

Since the metafile may create segments, the application must close any open segment before calling **GpiPlayMetaFile**. If the metafile creates segments, the function retains the segments only if the current drawing mode is DM_RETAIN or DM_DRAWANDRETAIN. If chained segments are retained, the function adds them to the end of the existing segment chain.

The **GpiPlayMetaFile** function can play a metafile any number of times.

Parameters

hps Identifies a presentation space.

hmf Identifies the metafile to play. It must have been created or loaded previously by using the **DevOpenDC** or **GpiLoadMetaFile** function.

cOptions Specifies the number of elements in the array pointed to by the *alOptions* parameter.

alOptions Points to the array specifying the load options. For a full description, see the following "Comments" section.

pcSegments Points to a variable for the count of renumbered segments. This parameter is reserved and is set to zero.

cchDesc Specifies the number of bytes in the buffer pointed to by the *pszDesc* parameter.

pszDesc Points to the buffer that receives the null-terminated string describing the metafile. This descriptive record is the record set by the **DevOpenDC** function for the metafile. If the buffer is smaller than the record, the function truncates the record.

Return Value

The return value is GPI_OK or GPI_HITS if the function is successful (it is GPI_HITS if the detectable attribute is set for the presentation space and a correlation hit occurs). The return value is GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following values:

PMERR_INCOMPATIBLE_METAFILE
PMERR_INV_ELEMENT_POINTER
PMERR_INV_HMF
PMERR_INV_HPS

PMERR_INV_IN_CURRENT_EDIT_MODE
PMERR_INV_LENGTH
PMERR_INV_LENGTH_OR_COUNT
PMERR_INV_METAFILE
PMERR_INV_MICROPS_ORDER
PMERR_INV_OUTSIDE_DRAW_MODE
PMERR_INV_PLAY_METAFILE_OPTION
PMERR_PROLOG_ERROR
PMERR_PS_BUSY
PMERR_STOP_DRAW_OCCURRED

Comments

The **GpiPlayMetaFile** function uses several options to control how a metafile is played. The options are specified in an array passed to the function by using the *alOptions* parameter. The array has at most ten elements, and there are eight predefined array indexes that can be used to access these elements. The following list describes the purpose and possible values for each element:

PMF_SEGBASE Specifies a reserved element. It must be zero.

PMF_LOADTYPE Specifies the transformation to use when playing the metafile. It can be one of the following values:

Value	Meaning
LT_DEFAULT	Default; same as LT_NOMODIFY.
LT_NOMODIFY	Use the current viewing transformation as set by the application by using the **GpiSetViewingTransformMatrix** function. This is the default action.
LT_ORIGINALVIEW	Use the viewing transformations defined in the metafile.

PMF_RESOLVE Specifies a reserved element. It must be RS_DEFAULT or RS_NODISCARD.

PMF_LCIDS Specifies whether to use tagged bitmaps and logical fonts from the metafile or from the application. It can be one of the following values:
metafile or from the application. It can be one of the following values:

Value	Meaning
LC_DEFAULT	Default; same as LC_NOLOAD.
LC_NOLOAD	Use the tagged bitmaps and logical fonts defined by the application. The application must define the appropriate objects and local identifiers before playing the metafile. This is the default.
LC_LOADDISC	Use the tagged bitmaps and logical fonts defined in the metafile. The function loads the object from the metafile and assigns a local identifier. If the local identifier is already defined by the application, the function deletes the identifier before creating the new object.

PMF_RESET Specifies whether the presentation space should be reset before playing the metafile, with the page units and size being set as defined in the metafile. It can be one of the following values:

Value	Meaning
RES_DEFAULT	Default; same as RES_NORESET.
RES_NORESET	Does not reset the presentation space.
RES_RESET	Resets the presentation space. The function resets the page units and page size to the values specified by the metafile. It then sets up default transformations, based on page units and size, as if the presentation space had just been created with these values, and modifies the device transformation (if necessary) to ensure that the physical size of the metafile picture is preserved. Finally, it resets the presentation space as if calling the **GpiResetPS** function with the GRES_ALL option.

PMF_SUPPRESS Specifies whether to continue playing the metafile after resetting the presentation space. It can be one of the following values:

Value	Meaning
SUP_DEFAULT	Default; same as SUP_NOSUPPRESS.
SUP_NOSUPPRESS	Does not suppress the metafile.
SUP_SUPPRESS	Suppresses the metafile after the presentation space is reset as specified by the PMF_RESET option. All other options are ignored.

PMF_COLORTABLES Specifies whether to use logical color tables from the metafile or from the application. It can be one of the following values:

Value	Meaning
CTAB_DEFAULT	Default; same as CTAB_NOMODIFY.
CTAB_NOMODIFY	Uses the logical color table defined by the application. This is the default.
CTAB_REPLACE	Uses the logical color tables implied by or given in the metafile. The application's existing logical color table is overwritten.

PMF_COLORREALIZABLE Specifies whether the logical color tables defined by the metafile should be realizable. It can be one of the following values:

Value	Meaning
CREA_DEFAULT	Default; same as CREA_NOREALIZE.
CREA_REALIZE	Creates realizable color tables.
CREA_NOREALIZE	Does not create realizable color tables. This is the default.

PMF_PATHBASE Specifies a reserved element. It must be zero.

PMF_RESOLVEPATH Specifies a reserved element. It must be RSP_DEFAULT or RSP_NODISCARD.

Example

This example uses the **GpiPlayMetaFile** function to play the given metafile. The function uses all the default actions for playing the metafile.

```
HMF hmf;
LONG cSegments;
CHAR szBuffer[80];

hmf = GpiLoadMetafile(hab, "sample.met");
GpiPlayMetafile(hps, hmf, 0L, NULL, &cSegments, 80L, szBuffer);
```

See Also

DevCloseDC, DevOpenDC, GpiCreateLogColorTable, GpiCreateLogFont, GpiLoadMetaFile, GpiResetPS, GpiSetDrawingMode, GpiSetViewing-TransformMatrix

Corrections

The default value for PMF_COLORREALIZABLE is CREA_NOREALIZE, not CREA_REALIZE.

■ GpiPolyLine — Correction

LONG GpiPolyLine(*hps, cptl, aptl* **)**

```
HPS hps;        /* presentation-space handle              */
LONG cptl;      /* number of points in array              */
PPOINTL aptl;   /* address of array of structures for points */
```

The **GpiPolyLine** function draws one or more straight lines. The function draws the lines by using the points specified by the *aptl* parameter. The function needs at least one point to draw a line. If a point is specified, the function draws the line from the current position to the point. For each additional line, the function needs exactly one more point, and uses the end point of the last line as the starting point for the next. The function draws the lines by using the current values of the line-color, line-mix, line-width, and line-type attributes.

The **GpiPolyLine** function moves the current position to the end point of the last line.

Parameters

hps Identifies a presentation space.

cptl Specifies the number of points. This parameter must be greater than or equal to zero.

aptl Points to an array of **POINTL** structures that contains the points. The **POINTL** structure has the following form:

```
typedef struct _POINTL {
    LONG  x;
    LONG  y;
} POINTL;
```

Return Value

The return value is GPI_OK or GPI_HITS if the function is successful (it is GPI_HITS if the detectable attribute is set for the presentation space and a correlation hit occurs). The return value is GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_INV_LENGTH_OR_COUNT
PMERR_PS_BUSY

Example

This example uses the **GpiPolyLine** function to draw a triangle:

```
POINTL ptlTriangle[] = { 100, 100, 200, 0, 0, 0 };

GpiMove(hps, &ptlTriangle[2]);          /* moves to end point (0, 0) */
GpiPolyLine(hps, 3L, &ptlTriangle[1]);  /* draws triangle            */
```

See Also

GpiLine, GpiMove, GpiSetAttrs, GpiSetColor, GpiSetCurrentPosition, GpiSetLineType

Corrections

The example did not draw a triangle because the starting point for **GpiPolyLine** was the same point as that moved to in the **GpiMove** function. The **GpiPolyLine** function was also missing a parameter.

■ GpiQueryBitmapBits — Correction

```
LONG GpiQueryBitmapBits( hps, lScanStart, cScan, pbBuffer, pbmi)
HPS hps;              /* presentation-space handle              */
LONG lScanStart;      /* number for first scan line to retrieve */
LONG cScan;           /* number of scan lines to retrieve       */
PBYTE pbBuffer;       /* address of buffer for bitmap image data */
PBITMAPINFO pbmi;     /* address of structure for bitmap info   */
```

The **GpiQueryBitmapBits** function copies image data from a bitmap to the buffer pointed to by the *pbBuffer* parameter. The function copies the image data from the bitmap currently set for the presentation space. The presentation space must be associated with a memory device context.

To copy the image data, the function needs the count of planes and adjacent color bits specified in the fields of the structure pointed to by the *pbmi* parameter. That is, the **cPlanes** and **cBitCount** fields must be set before you call the function. Also, the **cbFix** field must be set to 12. The function then copies the image data to the buffer. The buffer must have sufficient space to hold all the bytes of image data being copied. The number of bytes for the buffer is equal to the number of scan lines to copy, multiplied by the width of the bitmap in bytes (rounded up to the next multiple of 4), multiplied by the number of color planes. The width has to be a multiple of 4, since the function rounds the length of each scan line to a multiple of 4 bytes before copying. Also, the width must be multiplied by the number of adjacent color bits before rounding.

After copying the image data, the **GpiQueryBitmapBits** function fills the remaining fields in the structure pointed to by *pbmi*. These fields are the width and height of the bitmap and the array of RGB color values for the bitmap pels. An application must make sure there is sufficient space in the structure to receive all elements of the array of RGB color values. The number of elements in the array depends on the format of the bitmap.

Parameters

hps Identifies the presentation space.

lScanStart Specifies the number of the first scan line to copy to the buffer. If this parameter is zero, the function copies the first scan line in the bitmap.

cScan Specifies the number of scan lines to copy.

pbBuffer Points to the buffer that receives the bitmap image data. It must be large enough to hold all the bytes of the image data, from the scan line specified by the *lScanStart* parameter to the end of the bitmap.

pbmi Points to the **BITMAPINFO** structure that receives the bitmap information table. The **BITMAPINFO** structure has the following form:

```
typedef struct _BITMAPINFO {
    ULONG  cbFix;
    USHORT cx;
    USHORT cy;
    USHORT cPlanes;
    USHORT cBitCount;
    RGB    argbColor[1];
} BITMAPINFO;
```

Depending on the format of the given bitmap, an application may need to allocate extra bytes for the structure to hold the additional elements for the **argbColor** field.

Return Value

The return value is the number of scan lines retrieved if the function is successful or BMB_ERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INCORRECT_DC_TYPE
PMERR_INV_DC_TYPE
PMERR_INV_HPS
PMERR_INV_INFO_TABLE
PMERR_INV_LENGTH_OR_COUNT
PMERR_INV_SCAN_START
PMERR_NO_BITMAP_SELECTED
PMERR_PS_BUSY

Comments

If the requested color format is not the same as the bitmap's color format, the function converts the bitmap image data to the requested format.

For any scan line, the bits for the pixels are tightly packed, with the bits for the first pixel stored in the most significant bits of the first byte. If necessary, a scan line is padded at the end so that each scan line begins on a 32-bit boundary.

Example

This example uses **GpiQueryBitmapBits** to copy the image data of a bitmap from a presentation space associated with a memory device context.

```
BITMAPINFOHEADER bmp = { 12, 640, 350, 1, 1 };
LONG cbBuffer, cbBitmapInfo;
SEL selBuffer, selBitmapInfo;
PBYTE pbBuffer;
PBITMAPINFO pbmi;

/*
 * Compute the size of the image-data buffer and the bitmap
 * information structure.
 */

cbBuffer = (((bmp.cBitCount * bmp.cx) + 31) / 32)
    * 4 * bmp.cy * bmp.cPlanes;
cbBitmapInfo = sizeof(BITMAPINFO) +
    (sizeof(RGB) * (1 << bmp.cBitCount));

/*
 * Allocate memory for the image data-buffer and the bitmap
 * information structure.
 */

DosAllocSeg(cbBuffer, &selBuffer, SEG_NONSHARED);
pbBuffer = MAKEP(selBuffer, 0);
DosAllocSeg(cbBitmapInfo, &selBitmapInfo, SEG_NONSHARED);
pbmi = MAKEP(selBitmapInfo, 0);

/* Copy the image data. */

pbmi->cbFix = 12;
pbmi->cPlanes = 1;
pbmi->cBitCount = 1;
GpiQueryBitmapBits(hps, 0L, (LONG) bmp.cy, pbBuffer, pbmi);
```

See Also

GpiLoadBitmap, GpiQueryBitmapParameters, GpiSetBitmapBits

Corrections

The first bits in a scan line are stored in the most significant bits of the first byte of the scan line.

■ GpiQueryCharDirection — Change

LONG GpiQueryCharDirection(*hps* **)**
HPS *hps*; /* presentation-space handle */

The **GpiQueryCharDirection** function retrieves the current value of the character-direction attribute. This function cannot be used in an open segment when the drawing mode is DM_RETAIN.

Parameters

hps Identifies the presentation space.

Return Value

The return value is the current character-direction attribute if the function is successful, or CHDIRN_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HPS
PMERR_INV_IN_RETAIN_MODE
PMERR_PS_BUSY

Comments In MS OS/2, version 1.2, the following character directions are available:

Value	Meaning
CHDIRN_LEFTRIGHT	Left to right
CHDIRN_RIGHTLEFT	Right to left
CHDIRN_TOPBOTTOM	Top to bottom
CHDIRN_BOTTOMTOP	Bottom to top

See Also **GpiSetCharDirection, GpiSetDrawingMode**

Changes Character directions other than the default are allowed.

■ GpiQueryCharStringPos — Correction

```
BOOL GpiQueryCharStringPos( hps, flOptions, cchString, pchString, adx, aptl)
HPS hps;                /* presentation-space handle                    */
ULONG flOptions;        /* option flags                                 */
LONG cchString;         /* length of the string                         */
PCH pchString;          /* address of string to examine                 */
PLONG adx;              /* address of array for increment values        */
PPOINTL aptl;           /* address of array of structures for points    */
```

The **GpiQueryCharStringPos** function determines a position for each character in the string pointed to by the *pchString* parameter. Each position is the position of the character in world coordinates as if it were drawn by using the **GpiCharStringPos** function.

The **GpiQueryCharStringPos** function copies the character positions to the array of structures pointed to by the *aptl* parameter. It uses the current character attributes or the array of vector increments specified by the *adx* parameter to determine the positions. The function cannot be used in an open segment when the drawing mode is DM_RETAIN.

Parameters

hps Identifies the presentation space.

flOptions Specifies whether to use the vector increments specified by the *adx* parameter. It can be one of the following values:

Value	Meaning
0	Advances the current position after each character by using the width of the character. The *adx* parameter is ignored.
CHS_VECTOR	Advances the current position after each character by using the next value in the array *adx*. The current character direction defines the direction in which the current position is advanced.

cchString Specifies the length of the string pointed to by the *pchString* parameter.

pchString Points to the character string to examine.

adx Points to an array of increment values. Each value is a 4-byte signed integer specifying the distance (in world coordinates) to advance the current position for each character. There must be one value for each character in the string. The first element specifies the distance for the first character, the second element for the second character, and so on. This parameter may be NULL if the *flOptions* parameter is set to zero.

aptl Points to the array of **POINTL** structures that receives the position (in world coordinates) of each character in the string. The array must be large enough for each character in the string, plus one final point that contains the position of the first character that follows the string. The **POINTL** structure has the following form:

```
typedef struct _POINTL  {
    LONG  x;
    LONG  y;
} POINTL;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_CHAR_POS_OPTIONS
PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_INV_IN_RETAIN_MODE
PMERR_INV_LENGTH_OR_COUNT
PMERR_INV_RECT
PMERR_PS_BUSY

Example

This example calls the **GpiQueryCharStringPos** function to determine the location of each character in the string. Vector increments are not used.

```
CHAR szString[] = "Sample string";
POINTL aptl[sizeof(szString) + 1];

GpiQueryCharStringPos(hps,       /* presentation-space handle     */
    0L,                          /* does not use vector increments */
    sizeof(szString),            /* number of characters in string */
    szString,                    /* character string               */
    NULL,                        /* no vector increments           */
    aptl);                       /* array of structures for points */
```

See Also

GpiCharStringPos, GpiQueryCharStringPosAt, GpiSetDrawingMode

Corrections

The array of points specified in the *aptl* parameter must include not only a **POINTL** structure for each character in the string, but also one additional **POINTL** structure that will receive the position of the first character that follows the string.

■ GpiQueryDefArcParams New

```
BOOL GpiQueryDefArcParams(hps, parcp)
HPS hps;                 /* presentation-space handle           */
PARCPARAMS parcp;        /* pointer to structure for arc parameters */
```

The **GpiQueryDefArcParams** function retrieves the default arc parameters. The default arc parameters define the values given to the arc parameters of a presentation space whenever that presentation space is reset. (The arc parameters define the shape and orientation of the ellipses drawn using the arc functions.) A presentation space can be reset by using the **GpiResetPS** function.

Parameters

hps Identifies the presentation space.

parcp Points to the **ARCPARAMS** structure that receives the arc parameters. The **ARCPARAMS** structure has the following form:

```
typedef struct _ARCPARAMS {
    LONG lP;
    LONG lQ;
    LONG lR;
    LONG lS;
} ARCPARAMS;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_PS_BUSY

See Also

GpiQueryArcParams, GpiSetDefArcParams

■ GpiQueryDefAttrs New

```
BOOL GpiQueryDefAttrs(hps, lPrimType, flAttrMask, pbunAttrs)
HPS hps;                 /* presentation-space handle               */
LONG lPrimType;          /* primitive type                          */
ULONG flAttrMask;        /* attributes mask                         */
PBUNDLE pbunAttrs;       /* pointer to structure for default attributes */
```

The **GpiQueryDefAttrs** function retrieves the default attributes for a primitive. The default attributes define the values given to a presentation space's attributes when that presentation space is reset. The default attributes also define the value of attributes when they are explicitly set to the default by using the **GpiSetAttrs** function.

Parameters

hps Identifies the presentation space.

lPrimType Specifies which primitive type to retrieve attributes for. It can be one of the following values:

Value	Meaning
PRIM_AREA	Area primitives
PRIM_CHAR	Character primitives
PRIM_IMAGE	Image primitives
PRIM_LINE	Line and arc primitives
PRIM_MARKER	Marker primitives

flAttrMask Specifies which attributes to retrieve. The values for this parameter depend on the primitive type specified by the *lPrimType* parameter. This parameter can be any combination of the following values for a specific type:

Type	Values
PRIM_AREA	ABB_COLOR, ABB_BACK_COLOR, ABB_MIX_MODE, ABB_BACK_MIX_MODE, ABB_SET, ABB_SYMBOL, ABB_REF_POINT
PRIM_CHAR	CBB_COLOR, CBB_BACK_COLOR, CBB_MIX_MODE, CBB_BACK_MIX_MODE, CBB_SET, CBB_MODE, CBB_BOX, CBB_ANGLE, CBB_SHEAR, CBB_DIRECTION
PRIM_IMAGE	IBB_COLOR, IBB_BACK_COLOR, IBB_MIX_MODE, IBB_BACK_MIX_MODE
PRIM_LINE	LBB_COLOR, LBB_MIX_MODE, LBB_WIDTH, LBB_GEOM_WIDTH, LBB_TYPE, LBB_END, LBB_JOIN
PRIM_MARKER	MBB_COLOR, MBB_BACK_COLOR, MBB_MIX_MODE, MBB_BACK_MIX_MODE, MBB_SET, MBB_SYMBOL, MBB_BOX

If this parameter is zero, the function does not retrieve attributes but still returns a mask that specifies the attributes using default values.

pbunAttrs Points to the structure that receives the default attribute values for each attribute specified by the *flAttrMask* parameter. The type of structure depends on the value of the *lPrimType* parameter; it can be one of following structures:

Type	Structure
PRIM_AREA	**AREABUNDLE**
PRIM_CHAR	**CHARBUNDLE**
PRIM_IMAGE	**IMAGEBUNDLE**
PRIM_LINE	**LINEBUNDLE**
PRIM_MARKER	**MARKERBUNDLE**

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_HUGE_FONTS_NOT_SUPPORTED
PMERR_INV_BACKGROUND_COL_ATTR
PMERR_INV_CHAR_ANGLE_ATTR
PMERR_INV_CHAR_DIRECTION_ATTR
PMERR_INV_CHAR_MODE_ATTR
PMERR_INV_CHAR_SET_ATTR
PMERR_INV_CHAR_SHEAR_ATTR
PMERR_INV_COLOR_ATTR
PMERR_INV_COORDINATE
PMERR_INV_GEOM_LINE_WIDTH_ATTR
PMERR_INV_HPS
PMERR_INV_LINE_END_ATTR
PMERR_INV_LINE_JOIN_ATTR
PMERR_INV_LINE_TYPE_ATTR
PMERR_INV_LINE_WIDTH_ATTR
PMERR_INV_MARKER_SET_ATTR
PMERR_INV_MARKER_SYMBOL_ATTR
PMERR_INV_MIX_ATTR
PMERR_INV_PATTERN_ATTR
PMERR_INV_PATTERN_SET_ATTR
PMERR_INV_PATTERN_SET_FONT
PMERR_INV_PRIMITIVE_TYPE
PMERR_PS_BUSY
PMERR_UNSUPPORTED_ATTR
PMERR_UNSUPPORTED_ATTR_VALUE

See Also

GpiQueryAttrs, GpiSetDefAttrs

■ GpiQueryDefTag New

```
BOOL GpiQueryDefTag(hps, plTag)
HPS hps;        /* presentation-space handle */
PLONG plTag;    /* pointer to tag            */
```

The **GpiQueryDefTag** function retrieves the default primitive tag. A primitive tag is a way to identify a primitive stored in a segment.

Parameters

hps Identifies the presentation space.

plTag Points to the variable that receives the tag.

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HPS
PMERR_INV_MICROPS_FUNCTION
PMERR_PS_BUSY

See Also

GpiCorrelateChain, GpiCorrelateFrom, GpiSetDefTag

■ GpiQueryDefViewingLimits — New

```
BOOL GpiQueryDefViewingLimits( hps, prclLimits )
HPS hps;              /* presentation-space handle          */
PRECTL prclLimits;    /* pointer to structure for viewing limits */
```

The **GpiQueryDefViewingLimits** function retrieves the default viewing limits. The default viewing limits define the values given to a presentation space's viewing limits whenever that presentation space is reset. (The viewing limits specify a rectangle in model space that the system uses to clip output.) A presentation space can be reset by using the **GpiResetPS** function.

Parameters

hps Identifies the presentation space.

prclLimits Points to the **RECTL** structure that receives the coordinates of the default viewing limits. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_INV_VIEWING_LIMITS
PMERR_PS_BUSY

See Also

GpiQueryViewingLimits, GpiSetDefViewingLimits

■ GpiQueryFontFileDescriptions — Correction

```
LONG GpiQueryFontFileDescriptions( hab, pszFileName, pcFonts, pffdescs )
HAB hab;              /* anchor-block handle                     */
PSZ pszFileName;      /* address of the font-resource filename   */
PLONG pcFonts;        /* address of variable with number of fonts */
PFFDESCS pffdescs;    /* array of names                          */
```

The **GpiQueryFontFileDescriptions** function retrieves the typeface family and names contained in the specified file if the file is a font-resource file. The function copies the names to the array pointed to by the *pffdescs* parameter. Each name is a null-terminated string up to 32 characters long. The function copies all names in the file up to the number of names specified by the *pcFonts* parameter.

Parameters

hab Identifies the anchor block.

pszFileName Points to a null-terminated string. This string must be a valid MS OS/2 filename. If it does not specify a path and the *.fon* filename extension, the function appends the default extension (*.dll*) and looks for the font-resource file in the directories specified by the **libpath** command in the *config.sys* file.

pcFonts Points to a variable specifying the maximum number of typeface family and name pairs to retrieve. The function copies the actual number of descriptions it retrieved to this variable.

pffdescs Points to the array to receive the typeface family and names for each font. Each array element is itself a two-element array of type **FFDESCS**.

Return Value

The return value is the number of fonts for which details were not returned if the function is successful or GPI_ALTERROR if an error occurred.

Example

This example uses the **GpiQueryFontFileDescriptions** to retrieve the typeface family and names for the fonts in the *helv.dll* file. The function is called twice, once to determine the actual number of fonts in the file, and again to retrieve the descriptions.

```
PFFDESCS pffdescs;
SEL sel;
LONG cFonts = 0;

/* Retrieve a count of all fonts in the file. */

cFonts = GpiQueryFontFileDescriptions(hab, "helv", &cFonts, NULL);

/* Allocate space for the descriptions. */

DosAllocSeg((USHORT) (cFonts * sizeof(FFDESCS)), &sel, SEG_NONSHARED);
pffdescs = MAKEP(sel, 0);

/* Retrieve the descriptions. */

GpiQueryFontFileDescriptions(hab, "helv", &cFonts, pffdescs);
```

See Also

GpiQueryFonts

Corrections

In the example, the function retrieves information from the *helv.dll* file, not the *helv.fon* file. If no path and filename extension are given, the function by default searches for a file that has the *.dll* extension. Also, the cFonts variable must be set to zero for the first call to the function.

■ GpiQueryMetaFileBits Correction

BOOL GpiQueryMetaFileBits(*hmf*, *off*, *cbBuffer*, *pbBuffer* **)**

HMF *hmf*;	/* metafile handle */
LONG *off*;	/* offset to the first metafile byte to copy */
LONG *cbBuffer*;	/* length in bytes of buffer */
PBYTE *pbBuffer*;	/* address of buffer for metafile data */

The **GpiQueryMetaFileBits** function copies data from the metafile specified by *hmf* to the buffer pointed to by the *pbBuffer* parameter. The function copies the bytes of the metafile, up to the number of bytes specified by *cbBuffer*, starting at the byte whose offset from the beginning of the metafile is specified by the *off* parameter.

Parameters

hmf Identifies the memory metafile.

off Specifies the offset in bytes from the beginning of the metafile to the first byte to copy.

cbBuffer Specifies the number of bytes of metafile data to copy.

pbBuffer Points to the buffer to receive the metafile data. It must have the number of bytes specified by the *cbBuffer* parameter.

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HMF
PMERR_INV_METAFILE_LENGTH
PMERR_INV_METAFILE_OFFSET

Example

This example uses the **GpiQueryMetaFileBits** function to retrieve the graphics-order data from the specified metafile. The **GpiQueryMetaFileLength** function returns the length of the metafile.

```
HMF hmf;
LONG cBytes;
SEL sel;
LONG off;

hmf = GpiLoadMetaFile(hps, "sample.met");

/* Allocate the buffer for the metafile data. */

DosAllocSeg(0, &sel, SEG_NONSHARED);
pbBuffer = MAKEP(sel, 0);

cBytes = GpiQueryMetaFileLength(hmf);  /* gets length of metafile */

/* Retrieve up to 64K. */
GpiQueryMetaFileBits(
        hmf,          /* handle of metafile            */
        off,          /* offset of next byte to retrieve */
        65536L,       /* retrieves as much as possible   */
        pbBuffer);    /* buffer to receive metafile data */
```

See Also

GpiQueryMetaFileLength, GpiSetMetaFileBits

Corrections

In the example, the first parameter of the **GpiQueryMetaFileBits** function is a handle of the metafile, not a handle to the presentation space. Also, the metafile is assumed to be no greater than 64K. For larger metafiles, you can use the **DosAllocHuge** function to allocate segments to receive the metafile bits.

■ GpiResetPS — Change

BOOL GpiResetPS(*hps, flOption* **)**

```
HPS hps;           /* presentation-space handle */
ULONG flOption;    /* reset option              */
```

The **GpiResetPS** function resets the presentation space. In general, resetting the presentation space restores attributes to their default values—that is, the values given to the attributes when the presentation space was created or the values specified in the last call to the **GpiSetDefAttrs** function. The function can reset the presentation space in three ways: as if a segment were closed; as if the presentation space had just been created, but without deleting any resources; and as if the presentation space had just been created. It uses the *flOption* parameter to determine how to reset the presentation space.

The **GpiResetPS** function does not draw or erase the device. It is up to the application to erase the screen, if this is required. Also, the function does not affect the association between the specified presentation space and a device context.

The **GpiResetPS** function also deselects a bitmap if any are selected into a memory device context.

Parameters

hps Identifies the presentation space.

flOption Specifies the reset option. It can be one of the following:

Value	Meaning
GRES_ATTRS	Sets all current attributes to their default values, the current model transform to unity, and the current position to (0,0). The option also ends any open path, area, or element brackets and closes any open segment. Finally, it sets the current clip path and viewing limits to their widest possible values.
GRES_SEGMENTS	Resets as described for GRES_ATTRS, plus it deletes all retained segments, clears any boundary data, releases the clip region (if any), enables kerning (if the device supports it), and sets the default values for initial segment attributes, default viewing transform, graphics field, drawing mode, draw controls, edit mode, and attribute mode.
GRES_ALL	Resets as described for GRES_ATTRS and GRES_SEGMENTS, plus it deletes any logical fonts and local identifiers for bitmaps and sets the logical color table to its default value.

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HPS
PMERR_INV_RESET_OPTIONS
PMERR_PS_BUSY

See Also

GpiAssociate, GpiCreatePS, GpiSetAttrs

Changes

Calling **GpiResetPS** resets the attributes of the presentation space to the values it had when created or to the values specified in the last call to **GpiSetDefAttrs**.

Corrections

GpiResetPS also deselects a bitmap if any are selected into a memory device context.

■ GpiRotate New

```
BOOL GpiRotate( hps, pmatlf, flType, fxAngle, pptl)
HPS hps;             /* presentation-space handle            */
PMATRIXLF pmatlf;    /* pointer to structure with matrix     */
LONG flType;         /* transformation type                  */
FIXED fxAngle;       /* pointer to variable with rotation angle */
PPOINTL pptl;        /* pointer to structure with center point */
```

The **GpiRotate** function creates a transformation that can be used to rotate objects around a given point. **GpiRotate** either adds the specified rotation to an existing transformation or replaces the existing transformation with the rotation. The new transformation can be used in a subsequent call to any transformation function.

Parameters

hps Identifies the presentation space.

pmatlf Points to the **MATRIXLF** structure that contains the transformation matrix. The **MATRIXLF** structure has following form:

```
typedef struct _MATRIXLF {
    FIXED fxM11;
    FIXED fxM12;
    LONG  lM13;
    FIXED fxM21;
    FIXED fxM22;
    LONG  lM23;
    LONG  lM31;
    LONG  lM32;
    LONG  lM33;
} MATRIXLF;
```

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

flType Specifies how the specified matrix should be used to modify the transformation. It can be one of the following values:

Value	Meaning
TRANSFORM_ADD	Additive. The specified transformation matrix is combined with the existing transformation, with the existing transformation first, the new transformation second. This option is useful for incremental updates to transformations.
TRANSFORM_REPLACE	New/replace. The previous transformation is discarded and replaced by the specified transformation matrix.

fxAngle Specifies the rotation (in degrees) to use.

pptl Points to the **POINTL** structure that contains the coordinates of a point, relative to the origin, that defines the center of rotation. The **POINTL** structure has the following form:

```
typedef struct _POINTL {
    LONG  x;
    LONG  y;
} POINTL;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_INV_TRANSFORM_TYPE

See Also

GpiScale, GpiSetDefaultViewMatrix, GpiSetModelTransformMatrix, GpiSetSegmentTransformMatrix, GpiSetViewingTransformMatrix, GpiTranslate

■ GpiScale New

```
BOOL GpiScale( hps, pmatlf, flType, afxScale, pptl)
HPS hps;               /* presentation-space handle          */
PMATRIXLF pmatlf;      /* pointer to structure for matrix    */
LONG flType;           /* transformation type                */
PFIXED afxScale;       /* pointer to variable with scaling factor */
PPOINTL pptl;          /* pointer to structure with point data    */
```

The **GpiScale** function creates a transformation that can be used to scale (expand or contract) an object relative to a given point. **GpiScale** either adds the specified scaling factor to an existing transformation or replaces the existing transformation. The new transformation can be used in a subsequent call to any transformation function.

Parameters

hps Identifies the presentation space.

pmatlf Points to the **MATRIXLF** structure that contains the transformation matrix. The **MATRIXLF** structure has the following form:

```
typedef struct _MATRIXLF {
    FIXED fxM11;
    FIXED fxM12;
    LONG  lM13;
    FIXED fxM21;
    FIXED fxM22;
    LONG  lM23;
    LONG  lM31;
    LONG  lM32;
    LONG  lM33;
} MATRIXLF;
```

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

flType Specifies how a specified matrix should be used to modify the transformation. It can be one of the following values:

Value	Meaning
TRANSFORM_ADD	Additive. The specified transformation matrix is combined with the existing transformation, with the existing transformation first, the new transformation second. This option is useful for incremental updates to transformations.
TRANSFORM_REPLACE	New/replace. The previous transformation is discarded and replaced by the specified transformation matrix.

afxScale Points to the two-element array that contains the scaling factors to use. The first element specifies the scaling factor along the *x*-axis; the second specifies the scaling factor along the *y*-axis.

pptl Points to the **POINTL** structure that contains the coordinates of the point, relative to the origin, that defines the center of the scale. The **POINTL** structure has the following form:

```
typedef struct _POINTL {
    LONG  x;
    LONG  y;
} POINTL;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_INV_TRANSFORM_TYPE

See Also

GpiRotate, GpiSetDefaultViewMatrix, GpiSetModelTransformMatrix, GpiSetSegmentTransformMatrix, GpiSetViewingTransformMatrix, GpiTranslate

■ GpiSetCharDirection — Change

BOOL GpiSetCharDirection(*hps*, *flDirection* **)**

```
HPS hps;             /* presentation-space handle */
LONG flDirection;    /* character direction       */
```

The **GpiSetCharDirection** function sets the character direction for drawing characters. The character direction specifies the direction to advance after drawing a character, relative to the baseline.

If the attribute mode is AM_PRESERVE, the function saves the previous character direction on the attribute stack when it sets the new direction. The previous character direction can be retrieved by using the **GpiPop** function.

Parameters

hps Identifies the presentation space.

flDirection Specifies the character direction. This parameter can be one of the following values:

Value	Meaning
CHDIRN_DEFAULT	Default direction (left to right)
CHDIRN_LEFTRIGHT	Left to right
CHDIRN_RIGHTLEFT	Right to left
CHDIRN_TOPBOTTOM	Top to bottom
CHDIRN_BOTTOMTOP	Bottom to top

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_CHAR_DIRECTION_ATTR
PMERR_INV_HPS
PMERR_PS_BUSY

See Also

GpiPop, GpiQueryCharDirection, GpiSetAttrMode, GpiSetAttrs

Changes

The following character directions can now be specified for the *flDirection* parameter:

Value	Meaning
CHDIRN_DEFAULT	Default direction
CHDIRN_LEFTRIGHT	Left to right
CHDIRN_RIGHTLEFT	Right to left
CHDIRN_TOPBOTTOM	Top to bottom
CHDIRN_BOTTOMTOP	Bottom to top

■ GpiSetDefArcParams New

```
BOOL GpiSetDefArcParams( hps, parcp)
HPS hps;                  /* presentation-space handle              */
PARCPARAMS parcp;         /* pointer to structure with arc parameters */
```

The **GpiSetDefArcParams** function sets the default arc parameters. The default arc parameters define the values given to the arc parameters of a presentation space whenever that presentation space is reset. (The arc parameters define the shape and orientation of the ellipses drawn using the arc functions.) A presentation space can be reset using the **GpiResetPS** function.

Parameters

hps Identifies the presentation space.

parcp Points to the **ARCPARAMS** structure that contains the arc parameters. The **ARCPARAMS** structure has the following form:

```
typedef struct _ARCPARAMS {
    LONG lP;
    LONG lQ;
    LONG lR;
    LONG lS;
} ARCPARAMS;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_PS_BUSY

Comments Setting the default arc parameters does not immediately affect the arc parameters. The system uses the default arc parameters only when the presentation space is reset. The default arc parameters are reset when the presentation space is reset using the GRES_SEGMENT or GRES_ALL options of the **GpiResetPS** function. The reset values for the default arc parameters are lP=1, lQ=1, lR=0, and lS=0.

See Also **GpiFullArc, GpiPartialArc, GpiPointArc, GpiQueryDefArcParams**

■ GpiSetDefAttrs — New

```
BOOL GpiSetDefAttrs ( hps, lPrimType, flAttrMask, pbunAttrs )
HPS hps;              /* presentation-space handle                   */
LONG lPrimType;       /* primitive type                              */
ULONG flAttrMask;     /* attributes mask                             */
PBUNDLE pbunAttrs;    /* pointer to structure with default attributes */
```

The **GpiSetDefAttrs** function sets the default attributes for a primitive. The default attributes define the values given to the attributes of a presentation space when that presentation space is reset. The default attributes also define the value of attributes when they are explicitly set to the default using the **GpiSetAttrs** function.

Parameters

hps Identifies the presentation space.

lPrimType Specifies which primitive type to set default attributes for. It can be one of the following values:

Value	Meaning
PRIM_AREA	Area primitives
PRIM_CHAR	Character primitives
PRIM_IMAGE	Image primitives
PRIM_LINE	Line and arc primitives
PRIM_MARKER	Marker primitives

flAttrMask Specifies which default attributes to set. The values for this parameter depend on the primitive type specified by the *lPrimType* parameter. This parameter can be any combination of the following values for a specific type:

Type	Values
PRIM_AREA	ABB_COLOR, ABB_BACK_COLOR, ABB_MIX_MODE, ABB_BACK_MIX_MODE, ABB_SET, ABB_SYMBOL, ABB_REF_POINT
PRIM_CHAR	CBB_COLOR, CBB_BACK_COLOR, CBB_MIX_MODE, CBB_BACK_MIX_MODE, CBB_SET, CBB_MODE, CBB_BOX, CBB_ANGLE, CBB_SHEAR, CBB_DIRECTION

Type	Values
PRIM_IMAGE	IBB_COLOR, IBB_BACK_COLOR, IBB_MIX_MODE, IBB_BACK_MIX_MODE
PRIM_LINE	LBB_COLOR, LBB_MIX_MODE, LBB_WIDTH, LBB_GEOM_WIDTH, LBB_TYPE, LBB_END, LBB_JOIN
PRIM_MARKER	MBB_COLOR, MBB_BACK_COLOR, MBB_BACK_MIX_MODE, MBB_SET, MBB_SYMBOL, MBB_BOX, MBB_MIX_MODE

If this parameter is zero, no attributes are set, regardless of the value of the *pbunAttrs* parameter.

pbunAttrs Points to the buffer that contains attribute values for each default attribute specified by the *flAttrMask* parameter. The buffer format depends on the primitive type specified by the *lPrimType* parameter. The following structures can be used for the specified primitive types:

Type	Structure
PRIM_AREA	**AREABUNDLE**
PRIM_CHAR	**CHARBUNDLE**
PRIM_IMAGE	**IMAGEBUNDLE**
PRIM_LINE	**LINEBUNDLE**
PRIM_MARKER	**MARKERBUNDLE**

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_HUGE_FONTS_NOT_SUPPORTED
PMERR_INV_BACKGROUND_COL_ATTR
PMERR_INV_CHAR_ANGLE_ATTR
PMERR_INV_CHAR_DIRECTION_ATTR
PMERR_INV_CHAR_MODE_ATTR
PMERR_INV_CHAR_SET_ATTR
PMERR_INV_CHAR_SHEAR_ATTR
PMERR_INV_COLOR_ATTR
PMERR_INV_COORDINATE
PMERR_INV_GEOM_LINE_WIDTH_ATTR
PMERR_INV_HPS
PMERR_INV_LINE_END_ATTR
PMERR_INV_LINE_JOIN_ATTR
PMERR_INV_LINE_TYPE_ATTR
PMERR_INV_LINE_WIDTH_ATTR
PMERR_INV_MARKER_SET_ATTR
PMERR_INV_MARKER_SYMBOL_ATTR
PMERR_INV_MIX_ATTR
PMERR_INV_PATTERN_ATTR
PMERR_INV_PATTERN_SET_ATTR
PMERR_INV_PATTERN_SET_FONT

PMERR_INV_PRIMITIVE_TYPE
PMERR_PS_BUSY
PMERR_UNSUPPORTED_ATTR
PMERR_UNSUPPORTED_ATTR_VALUE

Comments Setting the default attributes for a primitive does not immediately affect the current attributes. The system uses the default attributes only when the presentation space is reset or when the **GpiSetAttrs** function is used to set the defaults. The default attributes are reset when the presentation space is reset using the GRES_SEGMENT or GRES_ALL options of the **GpiResetPS** function.

If an attempt is made to set an invalid default value, none of the specified default attribute values change. Some invalid default attribute values (for example, certain color and mix values), however, may not be detected until the attribute is used.

See Also **GpiQueryDefAttrs, GpiSetAttrs**

■ GpiSetDefTag New

```
BOOL GpiSetDefTag( hps, lTag)
HPS hps;        /* presentation-space handle */
LONG lTag;      /* tag                       */
```

The **GpiSetDefTag** function sets the default primitive tag. A primitive tag is a way to identify a primitive stored in a segment. This function sets the default primitive tag and the system applies this tag to all subsequent primitives.

Parameters

hps Identifies the presentation space.

lTag Specifies the tag. It must be an integer value.

Return Value The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_INV_HPS
PMERR_INV_MICROPS_FUNCTION
PMERR_PS_BUSY

See Also **GpiCorrelateChain, GpiCorrelateFrom, GpiCorrelateSegment, GpiQueryDefTag**

■ GpiSetDefViewingLimits New

```
BOOL GpiSetDefViewingLimits( hps, prclLimits)
HPS hps;              /* presentation-space handle               */
PRECTL prclLimits;    /* pointer to structure with viewing limits */
```

The **GpiSetDefViewingLimits** function sets the default viewing limits. The default viewing limits define the values given to the viewing limits of a presentation space whenever that presentation space is reset. (The viewing limits specify a rectangle in model space that the system uses to clip output.) A presentation space can be reset using the **GpiResetPS** function.

Parameters

hps Identifies the presentation space.

prclLimits Points to the **RECTL** structure that contains the coordinates of the default viewing limits. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_INV_VIEWING_LIMITS
PMERR_PS_BUSY

Comments

Setting the default viewing limits does not immediately affect the viewing-limits parameters. The system uses the default viewing limits only when the presentation space is reset. The default viewing limits are reset when the presentation space is reset using the GRES_SEGMENT or GRES_ALL options of the **GpiResetPS** function. The reset values for the default viewing limits are all of model space, meaning nothing is clipped.

See Also

GpiQueryDefViewingLimits, GpiSetViewingLimits

■ GpiSetPS — Correction

BOOL GpiSetPS(*hps*, *psizl*, *flOptions* **)**

HPS *hps*;	/* presentation-space handle */
PSIZEL *psizl*;	/* address of structure for presentation-space size */
ULONG *flOptions*;	/* options */

The **GpiSetPS** function sets the page size and units for the presentation space. This function is often used to change the device transformation for the presentation space.

Parameters

hps Identifies the presentation space.

psizl Points to the **SIZEL** structure that contains the size of the presentation space. The **SIZEL** structure has the following form:

```
typedef struct _SIZEL {
    LONG cx;
    LONG cy;
} SIZEL;
```

flOptions Specifies the presentation-space options. The options define the page unit for the presentation space. Although the *flOptions* parameter can include many other options (as specified by the **GpiCreatePS** function), the **GpiSetPS** function ignores all but the following options:

Option	Meaning
PU_ARBITRARY	Sets the page units to pels, but permits the units to be modified later by using the **GpiSetPageViewport** function.
PU_HIENGLISH	Sets the units to 0.001 inch.
PU_HIMETRIC	Sets the units to 0.01 millimeter.
PU_LOENGLISH	Sets the units to 0.01 inch.
PU_LOMETRIC	Sets the units to 0.1 millimeter.
PU_PELS	Sets the units to pels.
PU_TWIPS	Sets the units to 1/1440 inch (1/20 point).
GPIF_DEFAULT	Specifies that coordinates are stored as 4-byte integers (**LONG**). This value is the same as GPIF_LONG.
GPIF_SHORT	Specifies that coordinates are stored as 2-byte integers.
GPIF_LONG	Specifies that coordinates are stored as 4-byte integers.
PS_NORESET	Specifies that the presentation space cannot be fully reset, and that a reset equivalent to GRES_SEGMENTS is performed. (Otherwise, a full reset, equivalent to GRES_ALL, is performed.)

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_HDC
PMERR_INV_HPS
PMERR_INV_OR_INCOMPAT_OPTIONS
PMERR_INV_PS
PMERR_PS_BUSY

Comments

The **GpiSetPS** function does not affect the device context associated with the presentation space. This means the device context already associated remains associated. Also, the function does not change the type of presentation space. (Presentation-space types include the micro-presentation space and the normal presentation space.)

When this function is called, it resets the presentation space to a state that is equivalent to setting the value GRES_ALL in the **GpiResetPS** function.

See Also **GpiCreatePS, GpiResetPS**

Corrections **GpiSetPS** can be used to set the storage format for the presentation space by specifying one of the constants GPIF_DEFAULT, GPIF_LONG, or GPIF_SHORT. The PS_NORESET constant prevents the presentation space from being completely reset.

■ GpiSetViewingLimits — Correction

```
BOOL GpiSetViewingLimits( hps, prclLimits )
HPS hps;                /* presentation-space handle            */
PRECTL prclLimits;      /* address of structure with viewing limits */
```

The **GpiSetViewingLimits** function sets the viewing limits. The viewing limits specify a rectangle in model space that the system uses to clip output. The viewing limits include all points inside the rectangle and all points on the left and bottom edges, but do not include points on the right and top edges. Points on these edges are clipped.

The **GpiSetViewingLimits** function can be used in a segment to set the viewing limits for subsequent primitives in the segment. The viewing limits also apply to any called segments, unless the called segment itself sets the viewing limits.

Parameters

hps Identifies the presentation space.

prclLimits Points to the **RECTL** structure that contains the coordinates of the viewing limits. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

Return Value The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INV_COORDINATE
PMERR_INV_HPS
PMERR_INV_VIEWING_LIMITS
PMERR_PS_BUSY

Comments Unless the segments in the picture chain have the fast-chaining attribute, the system resets the default viewing limits when each segment in the chain is drawn. The default viewing limits include all model space—that is, nothing is clipped.

The segment and model transformations do not affect the viewing limits, but the viewing limits are affected by the current viewing and default viewing transformations.

If either the left boundary is greater than the right or the bottom boundary is greater than the top, a NULL rectangle is defined, and all points are clipped.

See Also **GpiQueryViewingLimits, GpiSetAttrMode**

Corrections If either the left boundary is greater than the right or the bottom boundary is greater than the top, a NULL rectangle is defined, and all points are clipped.

■ GpiTranslate **New**

```
BOOL GpiTranslate( hps, pmatlf, flType, pptl)
HPS hps;              /* presentation-space handle              */
PMATRIXLF pmatlf;     /* pointer to structure with matrix       */
LONG flType;          /* transformation type                    */
PPOINTL pptl;         /* pointer to structure with point data   */
```

The **GpiTranslate** function creates a transformation that can be used to translate (move) an object a specified direction and distance. **GpiTranslate** either adds the specified translation to an existing transformation or replaces the existing transformation. The new transformation can be used in a subsequent call to any transformation function.

Parameters

hps Identifies the presentation space.

pmatlf Points to the **MATRIXLF** structure that contains the transformation matrix. The **MATRIXLF** structure has the following form:

```
typedef struct _MATRIXLF {
    FIXED fxM11;
    FIXED fxM12;
    LONG  lM13;
    FIXED fxM21;
    FIXED fxM22;
    LONG  lM23;
    LONG  lM31;
    LONG  lM32;
    LONG  lM33;
} MATRIXLF;
```

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

flType Specifies how a specified matrix should be used to modify the transformation. It can be one of the following values:

Value	Meaning
TRANSFORM_ADD	Additive. The specified transformation matrix is combined with the existing transformation, with the existing transformation first, the new transformation second. This option is useful for incremental updates to transformations.
TRANSFORM_REPLACE	New/replace. The previous transformation is discarded and replaced by the specified transformation matrix.

pptl Points to the **POINTL** structure that contains the coordinates of a point, relative to the origin, that defines the required translation. The **POINTL** structure has the following form:

```
typedef struct _POINTL {
    LONG  x;
    LONG  y;
} POINTL;
```

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_INV_TRANSFORM_TYPE

See Also

GpiRotate, GpiScale, GpiSetDefaultViewMatrix, GpiSetModelTransform-Matrix, GpiSetSegmentTransformMatrix, GpiSetViewingTransformMatrix

■ GpiUnloadFonts — Correction

```
BOOL GpiUnloadFonts( hab, pszModName)
HAB hab;              /* anchor-block handle        */
PSZ pszModName;       /* address of the module name */
```

The **GpiUnloadFonts** function unloads font definitions that were previously loaded from the resource file specified by the *pszModName* parameter. Before unloading fonts, the application must delete any local identifiers previously assigned to the fonts. The function unloads the fonts for the application only. If any other applications have loaded the fonts, they remain available for those applications.

Parameters

hab Identifies the anchor block.

pszModName Points to a null-terminated string. This string must be a valid MS OS/2 filename. If it does not specify a path and the filename extension, the function appends the default extension (*.dll*) and searches for the font resource file in the directories specified by the **libpath** command in the *config.sys* file.

Return Value

The return value is GPI_OK if the function is successful or GPI_ERROR if an error occurred.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_FONT_FILE_NOT_LOADED
PMERR_FONT_NOT_LOADED
PMERR_OWN_SET_ID_REFS

See Also

GpiCreateLogFont, GpiDeleteSetId, GpiLoadFonts, GpiSetCharSet

Corrections

Before unloading fonts, the application *must* delete any local identifiers previously assigned to the fonts.

■ GpiWCBitBlt Correction

```
LONG GpiWCBitBlt( hps, hbm, cPoints, aptl, lRop, flOptions)
HPS hps;            /* presentation-space handle          */
HBITMAP hbm;        /* bitmap handle                      */
LONG cPoints;       /* number of points                   */
PPOINTL aptl;       /* address of structure with points   */
LONG lRop;          /* mixing function                    */
ULONG flOptions;    /* options                            */
```

The **GpiWCBitBlt** function copies a bitmap to a presentation space. It can also modify the bitmap within a rectangle in a presentation space. The exact operation carried out by **GpiWCBitBlt** depends on the raster operation specified by the *lRop* parameter.

If *lRop* directs **GpiWCBitBlt** to copy a bitmap, the function copies the bitmap specified by *hbm* to the presentation space. The presentation space must be associated with a device context for the display, for memory, or for some other suitable raster device. The *aptl* parameter points to an array of points that specify the corners of a rectangle in the bitmap as well as the corners of the rectangle in the presentation space to receive the bitmap. The bitmap rectangle is specified in device coordinates; the presentation-space rectangle in world coordinates. If the bitmap and presentation-space rectangles are not the same (after converting the presentation space to device coordinates), **GpiWCBitBlt** stretches or compresses the bitmap to fit the presentation-space rectangle.

If *lRop* directs **GpiWCBitBlt** to modify a bitmap, the function uses the raster operation to determine how to alter the bits in a rectangle in the presentation space. Raster operations include changes such as inverting existing bits, replacing bits with pattern bits, and mixing existing and pattern bits to create new colors. For some raster operations, the function mixes the bits of the bitmap with the presentation space and/or pattern bits.

Parameters

hps Identifies the presentation space.

hbm Identifies the bitmap.

cPoints Specifies the number of points pointed to by the *aptl* parameter. It must be 4.

aptl Points to an array of **POINTL** structures that contains the number of points specified in the *cPoints* parameter. The points must be given in the following order:

Element index	Coordinate
0	Specifies the lower-left corner of the target rectangle in world coordinates.
1	Specifies the upper-right corner of the target rectangle in world coordinates.
2	Specifies the lower-left corner of the source rectangle in device coordinates.
3	Specifies the upper-right corner of the source rectangle in device coordinates.

The **POINTL** structure has the following form:

```
typedef struct _POINTL  {
    LONG  x;
    LONG  y;
} POINTL;
```

lRop Specifies the raster operation for the function. It can be any value in the range 0 through 255 or one of the following values, which represent common raster operations:

Value	Meaning
ROP_DSTINVERT	Inverts the target.
ROP_MERGECOPY	Combines the source and the pattern using the bitwise AND operator.
ROP_MERGEPAINT	Combines the inverse of the source and the target using the bitwise OR operator.
ROP_NOTSRCCOPY	Copies the inverse of the source to the target.
ROP_NOTSRCERASE	Combines the inverse of the source and the inverse of the target bitmaps using the bitwise AND operator.
ROP_ONE	Sets all target pels to 1.
ROP_PATCOPY	Copies the pattern to the target.
ROP_PATINVERT	Combines the target and the pattern using the bitwise exclusive XOR operator.
ROP_PATPAINT	Combines the inverse of the source, the pattern, and target using the bitwise OR operator.
ROP_SRCAND	Combines the source and target bitmaps using the bitwise AND operator.
ROP_SRCCOPY	Copies the source bitmap to the target.
ROP_SRCERASE	Combines the source and the inverse of the target bitmaps using the bitwise AND operator.
ROP_SRCINVERT	Combines the source and target bitmaps using the bitwise exclusive OR operator.
ROP_SRCPAINT	Combines the source and target bitmaps using the bitwise OR operator.
ROP_ZERO	Sets all target pels to 0.

flOptions Specifies how to compress a bitmap if the target rectangle is smaller than the source. It can be one of the following values:

Value	Meaning
BBO_AND	Compresses two rows or columns into one by combining them with the bitwise AND operator. This value is useful for compressing bitmaps that have black images on a white background.
BBO_OR	Compresses two rows or columns into one by combining them with the bitwise OR operator. This value is the default and is useful for compressing bitmaps that have white images on a black background.

Value	Meaning
BBO_IGNORE	Compresses two rows or columns by throwing one out. This value is useful for compressing color bitmaps.

All values in the range 0x0100 to 0xFF00 are reserved for privately supported modes for particular devices.

Return Value

The return value is GPI_OK or GPI_HITS if the function is successful (it is GPI_HITS if the detectable attribute is set for the presentation space and a correlation hit occurs). The return value is GPI_ERROR if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_BASE_ERROR
PMERR_BITMAP_NOT_SELECTED
PMERR_INCOMPATIBLE_BITMAP
PMERR_INV_BITBLT_MIX
PMERR_INV_BITBLT_STYLE
PMERR_INV_COORDINATE
PMERR_INV_DC_TYPE
PMERR_INV_HBITMAP
PMERR_INV_HDC
PMERR_INV_IN_AREA
PMERR_INV_IN_PATH
PMERR_INV_LENGTH_OR_COUNT

Comments

The **GpiWCBitBlt** function can be used in an open segment. If the drawing mode is DM_DRAWANDRETAIN or DM_RETAIN, the function builds a graphics order in the current open segment. The order identifies the bitmap handle and uses uses long or short coordinates, as determined by the presentation-space format.

GpiWCBitBlt does not affect the pels in the upper and right boundaries of the presentation-space rectangle. This means the function draws up to but does not include those pels. Also, the function ignores any rotation transformations.

If the *lRop* parameter includes a pattern, **GpiWCBitBlt** uses the current area color, area background color, pattern set, and pattern symbol of the presentation space. Although the function may stretch or compress the bitmap, it never stretches or compresses the pattern.

If the presentation-space and the bitmap have different color formats, **GpiWCBitBlt** converts the bitmap color format as it copies the bitmap. This applies to bitmaps copied to a device context having a monochrome format. To convert a monochrome bitmap to a color bitmap, **GpiWCBitBlt** converts 1 pels to the presentation foreground color, and 0 pels to the current-area background color.

Example

This example uses **GpiWCBitBlt** to copy and compress a bitmap in a presentation space. The function copies the bitmap that is 100 pels wide and 100 pels high into a 50-by-50-pel rectangle at the location (300,400). Since the raster operation is ROP_SRCCOPY, **GpiWCBitBlt** replaces the image previously in the presentation-space rectangle. The function compresses the bitmap to fit the new rectangle by discarding extra rows and columns as specified by the BBO_IGNORE option.

```
HPS hps;
HBITMAP hbm;
POINTL aptl[4] = {
    300, 400,       /* lower-left corner of target              */
    350, 450,       /* upper-right corner of target             */
    0, 0,           /* lower-left corner of source              */
    100, 100 };     /* upper-right corner of source             */

GpiWCBitBlt(hps,    /* presentation space                       */
    hbm,            /* bitmap handle                            */
    4L,             /* four points needed to compress           */
    aptl,           /* points for source and target rectangles */
    ROP_SRCCOPY,    /* copy source replacing target             */
    BBO_IGNORE);    /* discard extra rows and columns           */
```

See Also **DevOpenDC, GpiBitBlt, GpiCreateBitmap, GpiLoadBitmap, GpiSetBitmap, GpiSetBitmapDimension, GpiSetBitmapId**

Corrections For the *aptl* parameter, the element indexes are 0, 1, 2, and 3. The array has at most four elements, not five.

■ HM_ACTIONBAR_COMMAND New

```
HM_ACTIONBAR_COMMAND
usCmd = (USHORT) SHORT1FROMMP(mp1);    /* command value */
```

The HM_ACTIONBAR_COMMAND message is sent when the user chooses a command from an application-supplied menu in the help window. The application should carry out the command identified by the *usCmd* parameter.

Parameters *usCmd* Low word of *mp1*. Specifies the command value.

Return Value An application should return zero if it processes this message.

Comments Applications can replace the menu in a help window by specifying a menu ID in the **HELPINIT** structure used when the help instance is created by using the **WinCreateHelpInstance** function. If an application replaces the menu, it receives the HM_ACTIONBAR_COMMAND message when the user chooses a command from the menu. Application-supplied menus should have command values in the range 0x7F00 through 0x7FFF.

See Also **WinCreateHelpInstance**

■ HM_CREATE_HELP_TABLE New

```
HM_CREATE_HELP_TABLE
mp1 = MPFROMP((PHELPTABLE) phtHelpTable); /* pointer to help table  */
mp2 = 0L;                                 /* not used, must be zero */
```

An application sends an HM_CREATE_HELP_TABLE message to a help window to set the help table for the help instance. The system uses the specified help table to locate help-panel IDs on subsequent requests for help.

Parameters *phtHelpTable* Low and high word of *mp1*. Points to the **HELPTABLE** structure that contains the help-table information. The **HELPTABLE** structure has the following form:

```
typedef struct _HELPTABLE {
    USHORT          idAppWindow;
    PHELPSUBTABLE   phstHelpSubTable;
    USHORT          idExtPanel;
} HELPTABLE;
```

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

Return Value The return value is FALSE.

Comments An application can use this message to replace the initial help table of a help instance or to set the table if no initial help table is given. The initial help table is specified in the **HELPINIT** structure used when the help instance is created by using the **WinCreateHelpInstance** function. This message replaces the help table without freeing any memory or resources associated with the initial help table.

The application must allocate space for the help table and fill the table with appropriate values before sending this message. The system does not check the validity of the help-table contents.

See Also **WinCreateHelpInstance**, HM_LOAD_HELP_TABLE

■ HM_DISMISS_WINDOW — New

```
HM_DISMISS_WINDOW
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an HM_DISMISS_WINDOW message to a help window to close the help window. Closing the help window does not destroy the help instance.

Parameters This message does not use any parameters.

Return Value The return value is FALSE if the help window is closed. It is TRUE if the help window was already closed.

Comments A help window is a modeless window. This means the user can view help and return to the application window without closing the help window. An application can use the HM_DISMISS_WINDOW message to close the help window if the user has not closed it.

See Also **WinDestroyHelpInstance**

■ HM_DISPLAY_HELP — New

```
HM_DISPLAY_HELP
mp1 = MPFROMP((PVOID) pHelpPanel);    /* panel ID or pointer to name */
mp2 = MPFROMSHORT((USHORT) usTypeFlag);  /* ID or name flag          */
```

An application sends an HM_DISPLAY_HELP message to a help window to display a specific help panel.

Parameters

pHelpPanel Low and high word of *mp1*. Points to a help-panel ID, points to a null-terminated help-panel name, or contains the help-panel ID in the low word and 0x0000 in the high word.

usTypeFlag Low word of *mp2*. Specifies whether the *pHelpPanel* parameter specifies a help-panel ID or name. The *usTypeFlag* parameter can be one of the following values:

Value	Meaning
HM_RESOURCEID	Specifies that *pHelpPanel* points to the help-panel ID or contains the help-panel ID in the low word.
HM_PANELNAME	Specifies that *pHelpPanel* points to the null-terminated help-panel name.

Return Value

The return value is FALSE if the help panel is displayed. Otherwise, it is an error value, which may be one of the following:

HMERR_DATABASE_NOT_OPEN
HMERR_PANEL_NOT_FOUND
HMERR_READ_LIB_FILE

Comments

The system searches for the specified panel in the help libraries opened for the help window and displays the first matching panel found.

See Also

HM_EXT_HELP, HM_HELP_CONTENTS, HM_HELP_INDEX, HM_KEYS_HELP

■ HM_ERROR — New

```
HM_ERROR
ulErrorCode = (ULONG) LONGFROMMP(mp1);    /* error type */
```

The HM_ERROR message is sent to notify an application of an error in a help window—errors that occur while the user views help. It does not notify the application of errors that result from messages sent by the application.

Parameters

ulErrorCode Low and high word of *mp1*. Specifies an error value, which may be one of the following:

HMERR_ALLOCATE_SEGMENT
HMERR_CLOSE_LIB_FILE
HMERR_CONTENT_NOT_FOUND
HMERR_DATABASE_NOT_OPEN
HMERR_FREE_MEMORY
HMERR_HELP_INSTANCE_UNDEFINE
HMERR_HELP_INST_CALLED_INVALID
HMERR_HELPITEM_NOT_FOUND
HMERR_HELPTABLE_UNDEFINE
HMERR_INDEX_NOT_FOUND
HMERR_INVALID_ASSOC_APP_WND
HMERR_INVALID_ASSOC_HELP_INST
HMERR_INVALID_DESTROY_HELP_INST
HMERR_INVALID_HELPSUBITEM_SIZE

HMERR_INVALID_HELP_INSTANCE_HDL
HMERR_INVALID_LIB_FILE
HMERR_INVALID_QUERY_APP_WND
HMERR_NO_FRAME_WND_IN_CHAIN
HMERR_NO_HELP_INST_IN_CHAIN
HMERR_NO_MEMORY
HMERR_OPEN_LIB_FILE
HMERR_PANEL_NOT_FOUND
HMERR_READ_LIB_FILE

Return Value An application should return zero if it processes this message.

Comments Because a help window does not display error messages to the user, the application should display its own messages when it receives an HM_ERROR message.

If an error occurs when creating the help instance using the **WinCreateHelpInstance** function, the system copies the error value to the **ulReturnCode** field in the **HELPINIT** structure used with **WinCreateHelpInstance.** If an error occurs for another function or for a message sent to a help window, the return value from the function or message sent specifies the error.

See Also **WinCreateHelpInstance**

■ HM_EXT_HELP — New

```
HM_EXT_HELP
mp1 = OL;    /* not used, must be zero */
mp2 = OL;    /* not used, must be zero */
```

An application sends an HM_EXT_HELP message to a help window to display the extended help panel.

Parameters This message does not use any parameters.

Return Value The return value is FALSE if the extended help panel is displayed. Otherwise, it is an error value, which may be one of the following:

HMERR_DATABASE_NOT_OPEN
HMERR_PANEL_NOT_FOUND
HMERR_READ_LIB_FILE

Comments For this message to display an extended help panel, the help table for the help instance must specify a help-panel ID that corresponds to the active window. (For example, the **idExtPanel** in the **HELPTABLE** structure used with the **WinCreateHelpInstance** function must be set to a valid help-panel ID.) If the help table specifies zero for the extended help-panel ID, the system sends the HM_EXT_HELP_UNDEFINED message to the application. In this case, the application should carry out some default action, such as displaying an error message or using the HM_DISPLAY_HELP message to display a help panel.

See Also HM_DISPLAY_HELP, HM_EXT_HELP_UNDEFINED, HM_HELP_INDEX, HM_KEYS_HELP

■ HM_EXT_HELP_UNDEFINED New

```
HM_EXT_HELP_UNDEFINED
```

The HM_EXT_HELP_UNDEFINED message notifies the application that an extended help panel is not defined for the active window.

Parameters This message does not use any parameters.

Return Value An application should return zero if it processes this message.

Comments The system displays extended help only if the help table for the help instance specifies a help-panel ID that corresponds to the active window. (For example, the **idExtPanel** in the **HELPTABLE** structure used with the **WinCreateHelp-Instance** function must be set to a valid help-panel ID.) If the help table specifies zero for the extended help-panel ID, the system sends the HM_EXT_HELP_UNDEFINED message to the application. In this case, the application should carry out some default action, such as displaying an error message or using the HM_DISPLAY_HELP message to display a help panel.

See Also HM_DISPLAY_HELP, HM_EXT_HELP

■ HM_HELP_CONTENTS New

```
HM_HELP_CONTENTS
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an HM_HELP_CONTENTS message to a help window to display the table of contents for the open help library.

Parameters This message does not use any parameters.

Return Value The return value is FALSE if the table of contents is displayed. Otherwise, it is an error value, which may be one of the following:

HMERR_DATABASE_NOT_OPEN
HMERR_PANEL_NOT_FOUND
HMERR_READ_LIB_FILE

See Also HM_DISPLAY_HELP, HM_HELP_INDEX, HM_KEYS_HELP

■ HM_HELP_INDEX New

```
HM_HELP_INDEX
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an HM_HELP_INDEX message to a help window to display the index for the open help library.

Parameters This message does not use any parameters.

Return Value The return value is FALSE if the index is displayed. Otherwise, it is an error value, which may be one of the following:

HMERR_DATABASE_NOT_OPEN
HMERR_PANEL_NOT_FOUND
HMERR_READ_LIB_FILE

See Also HM_DISPLAY_HELP, HM_HELP_CONTENTS, HM_KEYS_HELP

■ HM_HELPSUBITEM_NOT_FOUND New

```
HM_HELPSUBITEM_NOT_FOUND
usMode = (USHORT) SHORT1FROMMP(mp1);          /* help mode               */
idTopic = (USHORT) SHORT1FROMMP(mp2);         /* window ID for topic     */
idSubTopic = (USHORT) SHORT2FROMMP(mp2);      /* window ID for subtopic */
```

The HM_HELPSUBITEM_NOT_FOUND message notifies the application that the system failed to find a help panel in response to a user request for help.

Parameters *usMode* Low word of *mp1*. Specifies the context of the help request. This parameter can be one of the following values:

Value	Meaning
HLPM_FRAME	The help request is for a focus window that is a child window of the client window.
HLPM_MENU	The help request is for a selected menu item or submenu.
HLPM_WINDOW	The help request is for a focus window that is not a child window of the client window.

idTopic Low word of *mp2*. Specifies the ID of the active frame or dialog window or the submenu that contains the selection.

idSubTopic High word of *mp2*. Specifies the ID of the window that has the keyboard focus or the menu item that contains the selection.

Return Value An application should return FALSE to direct the system to display the extended help panel for the active window. An application should return TRUE to direct the system to do nothing.

Comments When an application receives this message, it should carry out a default action, such as displaying an error message or using the HM_DISPLAY_HELP message to display an explicitly specified help panel, or it can return FALSE to direct the system to display the extended help panel. If the application displays an error message or a help panel, it must return TRUE to prevent the system from displaying the extended help panel.

See Also HM_DISPLAY_HELP, HM_ERROR

■ HM_INFORM New

```
HM_INFORM
idPanel = (USHORT) SHORT1FROMMP(mp1);     /* help-panel ID */
```

The HM_INFORM message notifies an application that the user has chosen a hypertext field in the help window.

Parameters

idPanel Low word of *mp1*. Specifies the help-panel ID associated with the hypertext field.

Return Value

An application should return zero if it processes this message.

Comments

The system sends an HM_INFORM message only if the corresponding hypertext field was created using the **:inform** tag. The value of the *idPanel* parameter is the number specified with the tag. This is usually a help-panel ID, but it can be any number. When an application receives the HM_INFORM message, it can carry out any action; however, after the application returns from the message, the system displays the corresponding help panel if one exists.

See Also

HM_DISPLAY_HELP

■ HM_KEYS_HELP New

```
HM_KEYS_HELP
mp1 = OL;   /* not used, must be zero */
mp2 = OL;   /* not used, must be zero */
```

An application sends an HM_KEYS_HELP message to a help window to display the help panel that contains information about the application keys.

Parameters

This message does not use any parameters.

Return Value

The return value is FALSE if the keys-help panel is displayed. Otherwise, it is an error value, which may be one of the following:

HMERR_DATABASE_NOT_OPEN
HMERR_PANEL_NOT_FOUND
HMERR_READ_LIB_FILE

Comments

Because the keys-help-panel ID is not specified in the help table, the system sends an HM_QUERY_KEYS_HELP message to the window associated with the help window or to the active window. If the application returns the keys-help-panel ID, the system displays the keys-help window.

See Also

HM_DISPLAY_HELP, HM_EXT_HELP, HM_HELP_CONTENTS, HM_HELP_INDEX

HM_LOAD_HELP_TABLE New

```
HM_LOAD_HELP_TABLE
mp1 = MPFROM2SHORT(0xFFFF, (USHORT) idHelpTable);  /* help-table ID */
mp2 = MPFROMSHORT((USHORT) hmodModule);      /* module with resource */
```

An application sends an HM_LOAD_HELP_TABLE message to a help window to replace the existing help table (if any) with a help-table resource.

Parameters

idHelpTable Low word of *mp1*. Specifies the resource ID of the help-table resource.

hmodModule Low word of *mp2*. Identifies the module that contains the help-table resource.

Return Value The return value is FALSE.

Comments Applications can use this message to replace the initial help table of a help instance or to set the table if no initial help table is given. The initial help table is specified in the **HELPINIT** structure used when the help instance is created by using the **WinCreateHelpInstance** function. This message replaces the help table without freeing any memory or resources associated with the initial help table.

See Also **WinCreateHelpInstance**, HM_CREATE_HELP_TABLE

HM_QUERY_KEYS_HELP New

```
HM_QUERY_KEYS_HELP
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

The HM_QUERY_KEYS_HELP message is sent to an application to retrieve the keys-help-panel ID.

Parameters This message does not use any parameters.

Return Value An application should return the keys-help-panel ID. If no keys-help panel exists, the application should return an alternate panel ID, such as the ID for extended help.

Comments The system uses the returned ID to display the corresponding help panel. If the return value is not a valid help-panel ID, no help is displayed.

See Also HM_KEYS_HELP

HM_REPLACE_HELP_FOR_HELP New

```
HM_REPLACE_HELP_FOR_HELP
mp1 = MPFROMSHORT(idHelpForHelpPanel);   /* help-panel ID          */
mp2 = 0L;                                /* not used, must be zero */
```

An application sends an HM_REPLACE_HELP_FOR_HELP message to a help window to replace the general help panel (supplied by the system) with a specified help panel.

Parameters *idHelpForHelpPanel* Low word of *mp1*. Specifies a help-panel ID.

Return Value The return value is zero.

Comments A help window displays the general help panel whenever an application specifies zero for the help-panel ID in an HM_DISPLAY_HELP message. The general help panel is initially set by the system when the help instance is created; applications can replace the system-supplied help at any time. Applications that modify the help-window menu should also replace the general help information.

See Also HM_DISPLAY_HELP

HM_SET_ACTIVE_WINDOW New

```
HM_SET_ACTIVE_WINDOW
mp1 = MPFROMHWND(hwndActiveWindow);      /* active-window handle      */
mp2 = MPFROMHWND(hwndRelativeWindow);    /* application-window handle */
```

An application sends an HM_SET_ACTIVE_WINDOW message to a help window to set the active and relative windows. The active window is the window to which the system sends help messages. The relative window is the window next to which the system displays the help window.

Parameters *hwndActiveWindow* Low and high word of *mp1*. Identifies the active window. This value can be a window handle or NULL. If this parameter is NULL, the active and relative windows are determined by the system.

hwndRelativeWindow Low and high word of *mp2*. Identifies the relative window. This value can be a window handle or HWND_PARENT. If the value is HWND_PARENT, the system sets the relative window to be the parent window of the active window.

Return Value The return value is FALSE.

See Also **WinAssociateHelpInstance**

HM_SET_HELP_LIBRARY_NAME New

```
HM_SET_HELP_LIBRARY_NAME

mp1 = MPFROMP(pszHelpLibraryName); /* pointer to help-library name */
mp2 = OL;                          /* not used, must be zero       */
```

An application sends an HM_SET_HELP_LIBRARY_NAME message to Help Manager to identify the help library to search.

Parameters *pszHelpLibraryName* Low word of *mp1*. Points to the string that contains the help-library name used by Help Manager when it searches for the requested help topic.

Comments Sending an HM_SET_HELP_LIBRARY_NAME message replaces the current help library with the library specified.

■ HM_SET_HELP_WINDOW_TITLE New

```
HM_SET_HELP_WINDOW_TITLE
mp1 = MPFROMP(pszHelpWindowTitle);      /* pointer to new title   */
mp2 = 0L;                               /* not used, must be zero */
```

An application sends an HM_SET_HELP_WINDOW_TITLE message to a help window to change the window title.

Parameters *pszHelpWindowTitle* Low and high word of *mp1*. Points to the null-terminated string that contains the new Help-window title.

Return Value The return value is FALSE if the window title is set. Otherwise, it is an error value, which may be one of the following:

HMERR_ALLOCATE_SEGMENT
HMERR_NO_MEMORY

Comments The initial window title is specified by setting the **pszHelpWindowTitle** field in the **HELPINIT** structure used when the help instance is created by using the **WinCreateHelpInstance** function. The system allocates memory to save the title and frees the memory when the HM_SET_HELP_WINDOW_TITLE message is used to change the title.

See Also **WinCreateHelpInstance**

■ HM_SET_SHOW_PANEL_ID New

```
HM_SET_SHOW_PANEL_ID
mp1 = MPFROMSHORT(fVisible);    /* help-panel ID flag     */
mp2 = 0L;                       /* not used, must be zero */
```

An application sends an HM_SET_SHOW_PANEL_ID message to a help window to specify whether the window should display the help-panel ID along with the help panel title.

Parameters *fVisible* Low word of *mp1*. Specifies whether to display or hide the help-panel ID. This parameter can be one of the following values:

Value	Meaning
CMIC_HIDE_PANEL_ID	Turns off the show option. The help-panel ID is not displayed.
CMIC_SHOW_PANEL_ID	Turns on the show option. The help-panel ID is displayed.
CMIC_TOGGLE_PANEL_ID	Toggles the display of the help-panel ID.

Return Value The return value is zero.

Comments The help window displays the help-panel ID along with the help-panel title in the title bar of the help-panel window. The panel ID is enclosed in brackets.

Initially, an application specifies whether to display the help-panel ID by setting the **usShowPanelId** field in the **HELPINIT** structure when the help instance is created by using the **WinCreateHelpInstance** function.

See Also **WinCreateHelpInstance**

■ HM_TUTORIAL New

```
HM_TUTORIAL
pszTutorialName = (PSZ) PVOIDFROMMP(mp1);    /* pointer to tutorial */
```

The HM_TUTORIAL message is sent to a window when the user chooses the Tutorial command in the help window menu. The application can then invoke its own tutorial program.

Parameters

pszTutorialName Low and high word of *mp1*. Points to the null-terminated tutorial name.

Return Value

An application should return zero if it processes this message.

Comments

An application sets the name of the tutorial by setting the **pszTutorialName** field in the **HELPINIT** structure used when the help instance is created by using the **WinCreateHelpInstance** function. If a tutorial name is specified, the help window adds the Tutorial command to its Help menu.

See Also

WinCreateHelpInstance

■ KbdCharIn Change

```
USHORT KbdCharIn(pkbci, fWait, hkbd)
PKBDKEYINFO pkbci;    /* pointer to structure for keystroke info. */
USHORT fWait;         /* wait/no-wait flag                        */
HKBD hkbd;            /* keyboard handle                          */
```

The **KbdCharIn** function retrieves character and scan-code information from a logical keyboard. The function copies the information to a specified structure. 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.

Parameters

pkbci Points to the **KBDKEYINFO** structure that receives the keystroke information. The **KBDKEYINFO** structure has the following form:

```
typedef struct _KBDKEYINFO {
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

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

fWait Specifies whether to wait for keystroke information if none is available. If this parameter is IO_WAIT, the function waits for a keystroke if one is not available. If this parameter is IO_NOWAIT, the function returns immediately whether or not it retrieved any keystroke information. The **fbStatus** field in the **KBDKEYINFO** structure specifies whether a keystroke is received. The **fbStatus** field is nonzero if a keystroke is received or zero if not.

hkbd Identifies the logical keyboard. The handle must have been created by using the **KbdOpen** function.

Return Value

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

ERROR_KBD_FOCUS_REQUIRED
ERROR_KBD_INVALID_IOWAIT
ERROR_KBD_INVALID_HANDLE

Comments

The **KbdCharIn** function copies and removes keystroke information from the input buffer of the specified logical keyboard. Although echo mode for the logical keyboard may be turned on, **KbdCharIn** does not echo the characters it reads. If the keyboard is in ASCII mode, **KbdCharIn** retrieves keystroke information for each key pressed except shift keys. If the keyboard is in binary mode, **KbdCharIn** retrieves keystroke information for any key pressed except shift keys. In most cases, a shift key is pressed in combination with other keys to create a single keystroke. In binary mode with shift reporting turned on, a shift key by itself creates a keystroke this function can retrieve. For more information on binary mode and shift-reporting mode, see the **KbdSetStatus** function.

The **KbdCharIn** function retrieves extended ASCII codes, such as when the ALT key and another key, called the primary key, are pressed simultaneously. When the function retrieves an extended code, it sets the **chChar** field of the **KBDKEYINFO** structure to 0x0000 or 0x00E0. It also sets the **fbStatus** field to EXTENDED_CODE and copies the extended code to the **chScan** field. Note that both fields need to be examined to determine whether an extended code has been received. The extended code is usually the scan code of the primary key. In ASCII mode, the function retrieves only complete extended codes, which means that if both bytes of the extended code do not fit in the buffer, neither byte is retrieved. For more information, see the *Microsoft Operating System/2 Programmer's Reference, Volume 3.*

This function must be called twice to retrieve a code for a double-byte character set (DBCS). If the code retrieved is the first byte of a double-byte character, the **fbStatus** field of the **KBDKEYINFO** structure is set to 0x0080.

Restrictions

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

- It does not copy the system time to the **KBDKEYINFO** structure and there is no interim character support.
- It retrieves characters only from the default logical keyboard (handle 0).
- The **fbStatus** field can be 0x0000 or SHIFT_KEY_IN.
- The *hkbd* parameter is ignored.

Example

This example calls the **KbdCharIn** function to retrieve a character, and then displays the character on the screen:

```
KBDKEYINFO kbci;
KbdCharIn(&kbci,                    /* structure for data */
    IO_WAIT,                        /* waits for key      */
    0);                             /* keyboard handle    */
VioWrtTTY(&kbci.chChar, 1, 0);
```

See Also

KbdGetStatus, KbdOpen, KbdPeek, KbdSetStatus, KbdStringIn

Changes

In order to allow for input 0x00E0 as a normal character, a new value has been added to the **fbStatus** field of the **KBDKEYINFO** structure. In order to detect an extended code, both of the following conditions must be true:

- **chChar** must be equal to 0x0000 or 0x00E0
- **fbStatus** must be equal to EXTENDED_CHAR

■ KbdGetHWID New

```
USHORT KbdGetHWID(pkbdhwid, hkbd)
PKBDHWID pkbdhwid;    /* pointer to structure for ID number */
HKBD hkbd;            /* keyboard handle                    */
```

The **KbdGetHWID** function retrieves the hardware ID number of a keyboard.

Parameters

pkbdhwid Points to the **KBDHWID** structure that receives the ID number of the keyboard. The **KBDHWID** structure has the following form:

```
typedef struct _KBDHWID {
    USHORT cb;
    USHORT idKbd;
    USHORT usReserved1;
    USHORT usReserved2;
} KBDHWID;
```

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

hkbd Identifies the logical keyboard. This handle must have been created by using the **KbdOpen** function.

Return Value

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

ERROR_KBD_DETACHED
ERROR_KBD_INVALID_HANDLE
ERROR_KBD_PARAMETER

Example

This example opens a logical keyboard, and then calls the **KbdGetHWID** function to retrieve the hardware ID number of that keyboard:

```
HKBD hkbd;
KBDHWID kbhw;

KbdOpen(&hkbd);                   /* opens keyboard           */
KbdGetFocus(IO_WAIT, hkbd);       /* gets focus for keyboard */
kbhw.cb = sizeof(kbhw);           /* sets structure length   */
KbdGetHWID(&kbhw, hkbd);          /* gets ID number          */
```

See Also

DosDevIOCtl, KbdOpen

■ KbdRegister — Change

```
USHORT KbdRegister(pszModuleName, pszEntryName, fFunctions)
PSZ pszModuleName;      /* pointer to string for module name      */
PSZ pszEntryName;       /* pointer to string for entry-point name */
ULONG fFunctions;       /* function flags                         */
```

The **KbdRegister** function registers a **Kbd** subsystem for the specified logical keyboard. The function temporarily replaces the specified default **Kbd** functions with the functions in the specified module. Once **KbdRegister** replaces a function, MS OS/2 passes any subsequent call to the replaced function to a function in the given module. If a function is not replaced, MS OS/2 continues to call the default **Kbd** function.

Parameters

pszModuleName Points to the null-terminated string that contains the name of the dynamic-link module that specifies the replacement **Kbd** functions. The string must be a valid filename.

pszEntryName Points to the null-terminated string that contains the name of the dynamic-link entry-point function. For a full description, see the following "Comments" section.

fFunctions Specifies the flags for the functions to be replaced. This parameter can be any combination of the following values:

Value	Meaning
KR_KBDCHARIN	Replace **KbdCharIn.**
KR_KBDPEEK	Replace **KbdPeek.**
KR_KBDFLUSHBUFFER	Replace **KbdFlushBuffer.**
KR_KBDGETSTATUS	Replace **KbdGetStatus.**
KR_KBDSETSTATUS	Replace **KbdSetStatus.**
KR_KBDSTRINGIN	Replace **KbdStringIn.**
KR_KBDOPEN	Replace **KbdOpen.**
KR_KBDCLOSE	Replace **KbdClose.**
KR_KBDGETFOCUS	Replace **KbdGetFocus.**
KR_KBDFREEFOCUS	Replace **KbdFreeFocus.**
KR_KBDGETCP	Replace **KbdGetCp.**
KR_KBDSETCP	Replace **KbdSetCp.**
KR_KBDXLATE	Replace **KbdXlate.**
KR_KBDSETCUSTXT	Replace **KbdSetCustXt.**
KR_KBDGETHWID	Replace **KbdHWId.**

Return Value

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

ERROR_KBD_INVALID_ASCIIZ
ERROR_KBD_INVALID_MASK
ERROR_KBD_REGISTER

Comments

MS OS/2 passes a **Kbd** function to the given module by preparing the stack and calling the function pointed to by the *pszEntryName* parameter. The specified module must export the entry-point function name. The entry-point function must check the function code on the stack to determine which function is being requested and then pass control to the appropriate function in the module. The entry-point function can then access any additional parameters placed on the stack by the original call to **KbdRegister**.

Only one process in a screen group can use the **KbdRegister** function at any given time. That is, only one process can replace **Kbd** functions at any given time. The process can restore the default **Kbd** functions by calling the **Kbd-DeRegister** function. A process can replace **Kbd** functions any number of times, but only by first restoring the default functions and then reregistering the new functions.

The entry-point function (*FuncName*) must have the following form:

SHORT FAR *FuncName*(*selDataSeg*, *usReserved1*, *fFunction*,
ulReserved2, *usParam1*, *usParam2*, *usParam3*, *usParam4*,
usParam5, *usParam6*)
SEL *selDataSeg*;
USHORT *usReserved1*;
USHORT *fFunction*;
ULONG *ulReserved2*;
USHORT *usParam1*;
USHORT *usParam2*;
USHORT *usParam3*;
USHORT *usParam4*;
USHORT *usParam5*;
USHORT *usParam6*;

Parameters	Description
selDataSeg	Specifies the data-segment selector of the process that calls the specified **Kbd** function.
usReserved1	Specifies a reserved value that must not be changed. This value represents a return address for the MS OS/2 function that routes **Kbd** function calls.
fFunction	Specifies the function code of the function request. This parameter can be one of the following values:

Value	Meaning
0x0000	**KbdCharIn** called.
0x0001	**KbdPeek** called.
0x0002	**KbdFlushBuffer** called.
0x0003	**KbdGetStatus** called.
0x0004	**KbdSetStatus** called.
0x0005	**KbdStringIn** called.
0x0006	**KbdOpen** called.
0x0007	**KbdClose** called.
0x0008	**KbdGetFocus** called.
0x0009	**KbdFreeFocus** called.

Value	Meaning
0x000A	**KbdGetCp** called.
0x000B	**KbdSetCp** called.
0x000C	**KbdXlate** called.
0x000D	**KbdSetCustXt** called.
0x000E	**KbdGetHWId** called.

ulReserved2	Specifies a reserved value that must not be changed. This value represents the return address of the program that calls the specified **Kbd** function.
usParam1–usParam6	Specify up to six unsigned values passed with the call to the **Kbd** function. The number and type of parameters used depend on the specific function.

The entry-point function should determine which function is requested and then carry out an appropriate action by using the passed parameters. If necessary, the entry-point function can call a function within the same module to carry out the task. The entry-point or replacement function must leave the stack in the same state as it was received because the return addresses on the stack must be available in the correct order to return control to the program that originally called the **KbdRegister** function.

The registered function should return -1 to call the original function, 0 if no error occurred, or an error value.

In general, to access the keyboard the replacement function must use the input-and-output control functions for the keyboard.

The **KbdRegister** function itself cannot be replaced.

See Also **KbdDeRegister, KbdFlushBuffer**

Changes The KR_KBDGETTHWID constant for the new function **KbdGetHWId** has been added to the functions list and also to the return list.

■ MLM_CHARFROMLINE New

```
MLM_CHARFROMLINE
mp1 = MPFROMLONG((LINE) lLineNum);      /* line number           */
mp2 = 0L;                               /* not used, must be zero */
```

An application sends an MLM_CHARFROMLINE message to obtain the offset (number of characters from the beginning of the text) of the first character on the specified line in a multiple-line entry field (MLE).

Parameters *lLineNum* Low and high word of *mp1*. Specifies the line number. A value of zero specifies the first line. A value of -1 specifies the line that contains the cursor.

Return Value The return value is the 32-bit offset of the first character on the specified line.

Comments

If the *lLineNum* parameter specifies a line number greater than the line number of the last line of text in the MLE, the insertion point returned will be the point to the right of the last character in the MLE.

A line consists of all text up to a carriage return. A line may be displayed as several lines on the screen due to word-wrapping and still be considered a single line when specifying the line number for the *lLineNum* parameter.

Line numbers are zero-based. Therefore, the first line in an MLE is zero.

See Also

MLM_LINEFROMCHAR

■ MLM_CLEAR New

```
MLM_CLEAR
mp1 = 0L;      /* not used, must be zero */
mp2 = 0L;      /* not used, must be zero */
```

An application sends an MLM_CLEAR message to clear (delete) selected text in a multiple-line entry field (MLE).

Parameters

This message does not use any parameters.

Return Value

The return value is a 32-bit value (**ULONG**) that specifies the number of characters deleted.

See Also

MLM_CUT, MLM_DELETE

■ MLM_COPY New

```
MLM_COPY
mp1 = 0L;      /* not used, must be zero */
mp2 = 0L;      /* not used, must be zero */
```

An application sends an MLM_COPY message to copy selected multiple-line entry field (MLE) text to the clipboard.

Parameters

This message does not use any parameters.

Return Value

The return value is a 32-bit value (**ULONG**) that specifies the number of characters copied to the clipboard.

Comments

If no text is selected, the previous contents of the clipboard remain unaltered.

See Also

MLM_CUT, MLM_PASTE

■ MLM_CUT New

```
MLM_CUT
mp1 = OL;     /* not used, must be zero */
mp2 = OL;     /* not used, must be zero */
```

An application sends an MLM_CUT message to copy selected multiple-line entry-field (MLE) text to the clipboard and then clear the selected text.

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (**ULONG**) that specifies the number of characters copied and cleared.

Comments If no text is selected, the previous contents of the clipboard remain unaltered.

See Also MLM_COPY, MLM_DELETE, MLM_PASTE

■ MLM_DELETE New

```
MLM_DELETE
mp1 = MPFROMLONG((LINE) lBegin);     /* beginning of deletion */
mp2 = MPFROMLONG((ULONG) cch);       /* characters to delete  */
```

An application sends an MLM_DELETE message to delete the specified number of characters from a multiple-line entry field (MLE).

Parameters *lBegin* Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the first character to delete. If this parameter is set to -1, the current selection (if any) is deleted.

cch Low and high word of *mp2*. Specifies the number of characters to delete. This parameter is ignored if the *lBegin* parameter is set to -1.

Return Value The return value is a 32-bit value (**LONG**) that specifies the number of characters deleted.

See Also MLM_CUT

■ MLM_DISABLEREFRESH New

```
MLM_DISABLEREFRESH
mp1 = OL;     /* not used, must be zero */
mp2 = OL;     /* not used, must be zero */
```

An application sends an MLM_DISABLEREFRESH message to prevent repainting (refresh) of a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is always TRUE.

Comments When refresh is disabled, the MLE does not accept any keyboard or mouse input. If the mouse is moved over the MLE, it becomes an hourglass pointer.

See Also MLM_ENABLEREFRESH

■ MLM_ENABLEREFRESH New

```
MLM_ENABLEREFRESH
mp1 = 0L;     /* not used, must be zero */
mp2 = 0L;     /* not used, must be zero */
```

An application sends an MLM_ENABLEREFRESH message to enable repainting (refresh) of a multiple-line entry field (MLE). While the refresh state is enabled, the entire MLE window is repainted.

Parameters This message does not use any parameters.

Return Value The return value is always TRUE.

See Also MLM_DISABLEREFRESH

■ MLM_EXPORT New

```
MLM_EXPORT
mp1 = MPFROMP((PIPT) plOffset);      /* beginning of copy area */
mp2 = MPFROMP((PULONG) pcbCopy);     /* bytes to copy          */
```

An application sends an MLM_EXPORT message to export text from a multiple-line entry field (MLE) by copying the specified number of characters from the MLE to the buffer specified by the MLM_SETIMPORTEXPORT message. If all of the specified characters are on a single line, only the specified characters are copied. If the specified characters are on more than one line, the entire line containing the last specified character is copied.

Parameters

plOffset Low and high word of *mp1*. Points to the variable that specifies the offset (number of characters from the beginning of the text) of the first character to copy. A value of -1 specifies the current cursor position. On return, this variable contains the offset to the first character not copied to the buffer.

pcbCopy Low and high word of *mp2*. Points to the variable that specifies the number of characters to copy. On return, this variable is zero if the number of characters actually copied does not exceed the numbers specified to be copied. It is nonzero if the number of characters specified includes a line break and a portion of another line.

Return Value The return value is a 32-bit value (ULONG) that specifies the number of bytes actually copied. This value includes carriage-return and linefeed characters copied to the buffer.

Comments The text is copied in the form set by the MLM_FORMAT message. Note that the buffer is not zero-terminated.

All exports are done in full characters. Therefore, if the length of the buffer or the number of bytes to be exported results in the last byte transferred being only half of a double-byte character set (DBCS) character, the MLE does not transfer that byte.

See Also MLM_FORMAT, MLM_SETIMPORTEXPORT

■ MLM_FORMAT New

```
MLM_FORMAT
mp1 = MPFROMSHORT(usFormat);   /* format to set          */
mp2 = 0L;                      /* not used, must be zero */
```

An application sends an MLM_FORMAT message to set the format for importing to or exporting from a multiple-line entry field (MLE).

Parameters

usFormat Low word of *mp1*. Specifies the format to set. This parameter can be one of the following values:

Value	Meaning
MLFIE_CFTEXT	Specifies the clipboard text format. This format uses carriage-return/linefeed characters for line breaks on export, and recognizes linefeed, carriage-return/linefeed, or linefeed/carriage-return characters for line breaks on import. This is the default format.
MLFIE_NOTRANS	Specifies a format that uses linefeed characters for line breaks. This value guarantees that any text imported into the MLE in this form can be recovered in exactly the same form on export.
MLFIE_WINFMT	Specifies the format of the MLE window. This format recognizes carriage-return/linefeed characters for line breaks on import. It ignores the sequence carriage-return/carriage-return/linefeed. On export, it uses carriage-return/linefeed characters to denote a hard line break and carriage-return/carriage-return/linefeed characters to denote a soft line break caused by word-wrapping.

See Also

MLM_EXPORT, MLM_IMPORT, MLM_QUERYFORMATLINELENGTH, MLM_QUERYFORMATTEXTLENGTH

■ MLM_IMPORT New

```
MLM_IMPORT
mp1 = MPFROMP(plOffset);     /* import offset           */
mp2 = MPFROMLONG(cbCopy);    /* number of bytes to copy */
```

An application sends an MLM_IMPORT message to insert the contents of the buffer specified by the MLM_SETIMPORTEXPORT message into the multiple-line entry field (MLE).

Parameters

plOffset Low and high word of *mp1*. Points to the variable that specifies the offset (number of characters from the beginning of the text) to the edit-control buffer where the import buffer is to be inserted. A value of -1 specifies the current cursor position. On return, this variable contains the offset to the first character beyond the imported buffer.

cbCopy Low and high word of *mp2*. Specifies the number of bytes to import. If the last byte transferred is half of a double-byte character or part of a line-break sequence (carriage-return/linefeed), the last character is not transferred.

Return Value

The return value is a 32-bit value (**ULONG**) that specifies the number of bytes actually imported. This may be less than the value specified by the *cbCopy* parameter—if the last byte to copy included only part of a double-byte character or part of a line-break sequence. The return value is zero if the import would overflow the text limit set by the MLM_SETTEXTLIMIT message.

Comments

The contents of the buffer are interpreted as being in the form set by the MLM_FORMAT message.

See Also

MLM_FORMAT, MLM_SETIMPORTEXPORT, MLM_SETTEXTLIMIT, MLN_OVERFLOW, WM_CONTROL

■ MLM_INSERT New

```
MLM_INSERT
mp1 = MPFROMP(pszBuf);     /* pointer to text        */
mp2 = OL;                  /* not used, must be zero */
```

An application sends an MLM_INSERT message to insert text into a multiple-line entry field (MLE) at the current cursor position.

Parameters

pszBuf Low and high word of *mp1*. Points to the null-terminated string that contains the text to insert.

Return Value

The return value is TRUE if the text is inserted successfully or FALSE if an error occurs. If the inserted text overflows a text limit or format rectangle, an error occurs and an appropriate notification message is sent.

See Also

MLN_OVERFLOW, MLN_TEXTOVERFLOW, WM_CONTROL

■ MLM_LINEFROMCHAR New

```
MLM_LINEFROMCHAR
mp1 = MPFROMLONG(lOffset);    /* offset of MLE character */
mp2 = OL;                     /* not used, must be zero  */
```

An application sends an MLM_LINEFROMCHAR message to obtain the number of the line that contains the specified character in a multiple-line entry field (MLE).

Parameters

lOffset Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the specified character. A value of - 1 specifies that the number of the line that contains the cursor is returned. If the offset specified is greater than the total number of characters currently in the MLE, the number of the last line is returned.

Return Value

The return value is a 32-bit value (**ULONG**) that specifies the number of the line that contains the specified character.

Comments Line numbers are zero-based. Therefore, the first line in an MLE is zero.

See Also MLM_CHARFROMLINE

■ MLM_PASTE New

```
MLM_PASTE
mp1 = OL;    /* not used, must be zero */
mp2 = OL;    /* not used, must be zero */
```

An application sends an MLM_PASTE message to copy the contents of the clipboard to a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (**ULONG**) that specifies the number of characters copied. If the clipboard contains an incompatible format, the return value is zero.

See Also MLM_COPY, MLM_CUT

■ MLM_QUERYBACKCOLOR New

```
MLM_QUERYBACKCOLOR
mp1 = OL;    /* not used, must be 0 */
mp2 = OL;    /* not used, must be 0 */
```

An application sends an MLM_QUERYBACKCOLOR message to obtain background-color information for a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (**COLOR**) that specifies the background color. It can be one of the following values:

Value	Meaning
CLR_FALSE	All color planes are zeros.
CLR_TRUE	All color planes are ones.
CLR_DEFAULT	Default value; same as CLR_NEUTRAL.
CLR_WHITE	White.
CLR_BLACK	Black.
CLR_BACKGROUND	Reset color.
CLR_BLUE	Blue.
CLR_RED	Red.
CLR_PINK	Pink.
CLR_GREEN	Green.
CLR_CYAN	Cyan.
CLR_YELLOW	Yellow.

Value	Meaning
CLR_NEUTRAL	Neutral.
CLR_DARKGRAY	Dark gray.
CLR_DARKBLUE	Dark blue.
CLR_DARKRED	Dark red.
CLR_DARKPINK	Dark pink.
CLR_DARKGREEN	Dark green.
CLR_DARKCYAN	Dark cyan.
CLR_BROWN	Brown.
CLR_PALEGRAY	Light gray.

See Also MLM_QUERYTEXTCOLOR, MLM_SETBACKCOLOR

■ MLM_QUERYCHANGED New

```
MLM_QUERYCHANGED
mp1 = 0L;   /* not used, must be zero */
mp2 = 0L;   /* not used, must be zero */
```

An application sends an MLM_QUERYCHANGED message to determine if the text in a multiple-line entry field (MLE) has changed since the last time the changed flag was cleared.

Parameters This message does not use any parameters.

Return Value The return value is TRUE if the text has changed since the last time the changed flag was cleared. It is FALSE if the text is unchanged or if an error occurs.

Comments The changed flag can also be set or cleared by using an MLM_SETCHANGED message.

See Also MLM_SETCHANGED, MLN_CHANGE, WM_CONTROL

■ MLM_QUERYFIRSTCHAR New

```
MLM_QUERYFIRSTCHAR
mp1 = 0L;   /* not used, must be zero */
mp2 = 0L;   /* not used, must be zero */
```

An application sends an MLM_QUERYFIRSTCHAR message to retrieve the offset (number of characters from the beginning of the text) of the first visible character in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (ULONG) that specifies the offset of the first visible character.

See Also MLM_SETFIRSTCHAR

■ MLM_QUERYFONT New

```
MLM_QUERYFONT
mp1 = MPFROMP(pfattrs);     /* pointer to structure with font info. */
mp2 = OL;                   /* not used, must be zero               */
```

An application sends an MLM_QUERYFONT message to retrieve font information for a multiple-line entry field (MLE).

Parameters

pfattrs Low and high word of *mp1*. Points to the **FATTRS** structure that contains font information. The **FATTRS** structure has the following form:

```
typedef struct _FATTRS {
    USHORT  usRecordLength;
    USHORT  fsSelection;
    LONG    lMatch;
    CHAR    szFacename[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lMaxBaselineExt;
    LONG    lAveCharWidth;
    USHORT  fsType;
    USHORT  fsFontUse;
} FATTRS;
```

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

Return Value

The return value is TRUE if the system font is in use; otherwise, it is FALSE.

See Also

MLM_SETFONT

■ MLM_QUERYFORMATLINELENGTH New

```
MLM_QUERYFORMATLINELENGTH
mp1 = MPFROMLONG((LONG) lOffset); /* offset of beginning character */
mp2 = OL;                         /* not used, must be zero        */
```

An application sends an MLM_QUERYFORMATLINELENGTH message to retrieve the length (in bytes) of a line in a multiple-line entry field (MLE).

Parameters

lOffset Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the first character to count. If this value is -1, the current cursor position is used as the starting character.

Return Value

The return value is a 32-bit value (ULONG) that specifies the number of bytes between the specified character and the beginning of the next line. If the specified character is on the last line in the MLE, the number of bytes to the end of that line is returned.

Comments

The number of bytes returned for the end-of-line character is determined by the format specified by the MLM_FORMAT message. This format can be one of the following values:

Format	Description
MLFIE_CFTEXT	The end-of-line character is formatted as carriage-return/linefeed characters (2 bytes).
MLFIE_NOTRANS	The end-of-line character is formatted as a linefeed character (1 byte).

Format	Description
MLFIE_WINFMT	The end-of-line character for hard line breaks is formatted as carriage-return/linefeed characters (2 bytes). The end-of-line character for soft line breaks (line breaks caused by word-wrapping) is formatted as carriage-return/carriage-return/ linefeed characters (3 bytes).

See Also MLM_FORMAT, MLM_QUERYFORMATTEXTLENGTH, MLM_QUERYLINELENGTH

MLM_QUERYFORMATRECT New

```
MLM_QUERYFORMATRECT
mp1 = MPFROMP((PMLEFORMATRECT) pmlefrmrcl); /* point to MLEFORMATRECT */
mp2 = MPFROMP((PULONG) pflOptions);         /* point to variable      */
```

An application sends an MLM_QUERYFORMATRECT message to retrieve the dimensions used to define the format rectangle for a multiple-line entry field (MLE).

Parameters *pmlefrmrcl* Low and high word of *mp1*. Points to the **MLEFORMATRECT** structure that receives the format-rectangle dimensions for the MLE. The **MLEFORMATRECT** structure has the following form:

```
typedef struct _MLEFORMATRECT {
    LONG cxFormat;
    LONG cyFormat;
} MLEFORMATRECT;
```

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

pflOptions Low and high word of *mp2*. Points to the variable that receives the flags that specify how the format rectangle is to be used. A value of zero causes the MLE to remove any format rectangle and to ignore the *pmlefrmrcl* parameter. Otherwise, this parameter can be a combination of the following values:

Value	Meaning
MLFFMTRECT_LIMITHORZ	Specifies that the text within the MLE cannot exceed the horizontal dimension specified by the *pmlefrmrcl* parameter. If word-wrap mode is turned on before the format rectangle is set, lines automatically wrap to stay within the horizontal limit of the format rectangle. If word-wrap mode is turned off before the format rectangle is set, an MLN_PIXHORZOVERFLOW notification message is sent to the application whenever an operation would exceed the horizontal limit specified in the format rectangle.

Value	Meaning
MLFFMTRECT_LIMITVERT	Specifies that the text within the MLE cannot exceed the vertical dimension specified by the *pmlefrmrcl* parameter. An MLN_PIXVERTOVERFLOW notification message is sent to the application whenever an MLE operation would cause text to exceed the vertical limit.
MLFFMTRECT_MATCHWINDOW	Specifies that the format rectangle is to be kept the same size as the MLE window (minus the border or scroll bars).
MLFFMTRECT_FORMATRECT	Specifies that the format rectangle is to be kept the same size as the MLE window (minus the border or scroll bars) and that text cannot exceed the size of the window. This value is equivalent to combining the MLFFMTRECT_LIMITHORZ, MLFFMTRECT_LIMITVERT, and MLFFMTRECT_MATCHWINDOW values.

Return Value The return value is always FALSE.

See Also MLM_SETFORMATRECT

■ MLM_QUERYFORMATTEXTLENGTH New

```
MLM_QUERYFORMATTEXTLENGTH
mp1 = MPFROMLONG((LONG) lOffset);    /* offset of starting character */
mp2 = MPFROMLONG((LONG) cbChar);     /* characters to scan           */
```

An application sends an MLM_QUERYFORMATTEXTLENGTH message to retrieve the length (in bytes) of a range of characters in multiple-line entry field (MLE).

Parameters

lOffset Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the first character to count. If this parameter is set to -1, the current cursor position is used as the starting character.

cbChar Low and high word of *mp2*. Specifies the number of characters to scan. If this parameter is set to -1, the entire text is scanned.

Return Value The return value is a 32-bit value (**ULONG**) that specifies the number of bytes in the specified range of characters.

Comments The number of bytes returned for any end-of-line characters is determined by the format specified by the MLM_FORMAT message. This format can be one of the following values:

Format	Description
MLFIE_CFTEXT	The end-of-line character is formatted as carriage-return/linefeed characters (2 bytes).

Format	Description
MLFIE_NOTRANS	The end-of-line character is formatted as a linefeed character (1 byte).
MLFIE_WINFMT	The end-of-line character for hard line breaks is formatted as carriage-return/linefeed characters (2 bytes). The end-of-line character for soft line breaks (line breaks caused by word-wrapping) is formatted as carriage-return/carriage-return/linefeed characters (3 bytes).

See Also MLM_FORMAT

■ MLM_QUERYIMPORTEXPORT New

```
MLM_QUERYIMPORTEXPORT
mp1 = MPFROMP((PBYTE FAR *) ppBuf);     /* pointer to buffer      */
mp2 = MPFROMP((PUSHORT) pcbBuf);        /* pointer to buffer size */
```

An application sends an MLM_QUERYIMPORTEXPORT message to determine the address and size of the buffer used by the import/export buffer of a multiple-line entry field (MLE). The buffer must have been set previously by sending an MLM_SETIMPORTEXPORT message (or the returned parameters will be invalid).

Parameters *ppBuf* Low and high word of *mp1*. Points to the variable that receives the address of the import/export buffer.

pcbBuf Low word of *mp2*. Points to the variable that receives the size of the buffer pointed to by *ppBuf*.

Return Value The return value is always TRUE.

Comments The import/export buffer can be used to import to and export text from the MLE by using the MLM_IMPORT and MLM_EXPORT messages.

See Also MLM_EXPORT, MLM_IMPORT, MLM_SETIMPORTEXPORT

■ MLM_QUERYLINECOUNT New

```
MLM_QUERYLINECOUNT
mp1 = 0L;   /* not used, must be zero */
mp2 = 0L;   /* not used, must be zero */
```

An application sends an MLM_QUERYLINECOUNT message to retrieve the number of lines in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (ULONG) that specifies the number of lines in the MLE.

See Also MLM_QUERYTEXTLENGTH

■ MLM_QUERYLINELENGTH New

```
MLM_QUERYLINELENGTH
mp1 = MPFROMLONG(lOffset);     /* beginning of count     */
mp2 = OL;                      /* not used, must be zero */
```

An application sends an MLM_QUERYLINELENGTH message to retrieve the number of characters between the specified character and the beginning of the next line in a multiple-line entry control (MLE).

Parameters *lOffset* Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the first character to count. If this parameter is set to - 1, the current cursor position is used as the starting character.

Return Value The return value is a 32-bit value (ULONG) that specifies the number of characters between the specified character and the beginning of the next line. If the specified character is on the last line of the MLE, the number of characters to the end of that line is returned.

Comments The line break at the end of the line is counted as a single character.

See Also MLM_QUERYTEXTLENGTH

■ MLM_QUERYREADONLY New

```
MLM_QUERYREADONLY
mp1 = OL;   /* not used, must be zero */
mp2 = OL;   /* not used, must be zero */
```

An application sends an MLM_QUERYREADONLY message to determine whether the multiple-line entry field (MLE) is in read-only mode. While read-only mode is set, the user cannot change the contents of the text in the MLE.

Parameters This message does not use any parameters.

Return Value The return value is the read-only state of the MLE. The return value is TRUE when read-only mode is set.

See Also MLM_SETREADONLY

■ MLM_QUERYSEL New

```
MLM_QUERYSEL
mp1 = MPFROMSHORT(usQueryMode);    /* specifies the type of query */
mp2 = OL;                          /* not used, must be zero      */
```

An application sends an MLM_QUERYSEL message to retrieve the offsets (number of characters from the beginning of the text) of the characters selected in a multiple-line entry field (MLE).

Parameters

usQueryMode Low word of *mp1*. Specifies which offset to return. This parameter can be one of the following values:

Value	Meaning
MLFQS_MINMAXSEL	Returns the offsets of the selection in a single 32-bit value. The high word contains the offset of the ending selection character, and the low word will contain the offset of the beginning character. These values are invalid if the selection contains offsets greater than 64K.
MLFQS_MINSEL	Returns the minimum (leftmost) offset of the selection.
MLFQS_MAXSEL	Returns the maximum (rightmost) offset of the selection.
MLFQS_ANCHORSEL	Returns the offset of the first selected character.
MLFQS_CURSORSEL	Returns the offset of the cursor.

Return Value

The return value is a 32-bit value; its meaning depends on the setting of the *usQueryMode* parameter.

Example

This example sends two MLM_QUERYSEL messages to obtain the beginning and ending points of the current selection, sends an MLM_SETIMPORTEXPORT message to set up the export buffer, and then sends an MLM_EXPORT message to export the selection into the buffer.

```
LONG lStart, cch;
CHAR szBuf[500];

lStart = (LONG) WinSendMsg(hwndMle, MLM_QUERYSEL,
    (MPARAM) MLFQS_MINSEL, (MPARAM) OL);
cch = lStart - (LONG) WinSendMsg(hwndMle, MLM_QUERYSEL,
    (MPARAM) MLFQS_MAXSEL, (MPARAM) OL);
WinSendMsg(hwndMle, MLM_SETIMPORTEXPORT,
    (MPARAM) szBuf, (MPARAM) sizeof(szBuf));
WinSendMsg(hwndMle, MLM_EXPORT, (MPARAM) &lStart, (MPARAM) &cch);
```

See Also

MLM_EXPORT, MLM_QUERYSELTEXT, MLM_SETIMPORTEXPORT, MLM_SETSEL

MLM_QUERYSELTEXT New

```
MLM_QUERYSELTEXT
mp1 = MPFROMP((PCH) pchBuf);     /* pointer to buffer for selection */
mp2 = OL;                        /* not used, must be zero          */
```

An application sends an MLM_QUERYSELTEXT message to copy the selection from a multiple-line entry field (MLE) into the specified buffer.

Parameters

pchBuf Low and high word of *mp1*. Points to the buffer that receives the selected text.

Return Value

The return value is a 32-bit value (**ULONG**) that specifies the number of bytes actually placed in the buffer.

Comments The application must ensure that the selected text does not overflow the buffer. An application can send an MLM_QUERYSEL message to retrieve character offsets of the selection, and then send an MLM_QUERYFORMATTEXTLENGTH message to determine the number of bytes the selected text occupies.

See Also MLM_QUERYFORMATTEXTLENGTH, MLM_QUERYSEL

I MLM_QUERYTABSTOP New

```
MLM_QUERYTABSTOP
mp1 = OL;    /* not used, must be zero */
mp2 = OL;    /* not used, must be zero */
```

An application sends an MLM_QUERYTABSTOP message to retrieve the interval (in pels) at which tab stops are set in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 16-bit value (**USHORT**) that specifies the tab-stop interval.

See Also MLM_SETTABSTOP

I MLM_QUERYTEXTCOLOR New

```
MLM_QUERYTEXTCOLOR
mp1 = OL;    /* not used, must be zero */
mp2 = OL;    /* not used, must be zero */
```

An application sends an MLM_QUERYTEXTCOLOR message to obtain the color of the text in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value that indicates the color of the text. It can be one of the following values:

Value	Meaning
CLR_FALSE	All color planes are zeros.
CLR_TRUE	All color planes are ones.
CLR_DEFAULT	Default value; same as CLR_NEUTRAL.
CLR_WHITE	White.
CLR_BLACK	Black.
CLR_BACKGROUND	Reset color.
CLR_BLUE	Blue.
CLR_RED	Red.
CLR_PINK	Pink.
CLR_GREEN	Green.

Value	Meaning
CLR_CYAN	Cyan.
CLR_YELLOW	Yellow.
CLR_NEUTRAL	Neutral.
CLR_DARKGRAY	Dark gray.
CLR_DARKBLUE	Dark blue.
CLR_DARKRED	Dark red.
CLR_DARKPINK	Dark pink.
CLR_DARKGREEN	Dark green.
CLR_DARKCYAN	Dark cyan.
CLR_BROWN	Brown.
CLR_PALEGRAY	Light gray.

See Also MLM_SETTEXTCOLOR

■ MLM_QUERYTEXTLENGTH New

```
MLM_QUERYTEXTLENGTH
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an MLM_QUERYTEXTLENGTH message to retrieve the number of bytes in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (**LONG**) that specifies the number of bytes in the MLE. This value includes carriage-return and linefeed characters.

See Also MLM_QUERYFORMATTEXTLENGTH

■ MLM_QUERYTEXTLIMIT New

```
MLM_QUERYTEXTLIMIT
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an MLM_QUERYTEXTLIMIT message to retrieve the number of characters currently allowed in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value (**LONG**) that specifies the maximum number of characters currently allowed in the MLE. A return value of -1 indicates an unlimited number of characters are allowed.

See Also MLM_SETTEXTLIMIT

■ MLM_QUERYUNDO New

```
MLM_QUERYUNDO
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an MLM_QUERYUNDO message to determine if a multiple-line entry-field (MLE) operation can be undone.

Parameters This message does not use any parameters.

Return Value The return value is a 32-bit value that indicates whether an MLE operation can be undone and, if so, which message can be undone. The low word contains TRUE if the message can be undone or FALSE if the message was just undone. The high word contains the message, or it contains zero if no message is available to be undone. The following messages can be returned:

Message	Description
MLM_CLEAR	Indicates that the last MLM_CLEAR or MLM_DELETE message can be undone.
MLM_CUT	Indicates that the last MLM_CUT message can be undone.
MLM_INSERT	Indicates that the last MLM_INSERT message can be undone.
MLM_PASTE	Indicates that the last MLM_PASTE message can be undone.
MLM_SETFONT	Indicates that the last MLM_SETFONT message can be undone.
MLM_SETTEXTCOLOR	Indicates the last MLM_SETBACKCOLOR or MLM_SETTEXTCOLOR message can be undone.
WM_CHAR	Indicates that the last character entered by the user can be undone.

See Also MLM_RESETUNDO, MLM_UNDO

■ MLM_QUERYWRAP New

```
MLM_QUERYWRAP
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an MLM_QUERYWRAP message to retrieve the current state of word-wrapping in a multiple-line entry field (MLE).

Parameters This message does not use any parameters.

Return Value The return value is TRUE if word-wrapping is currently set. It is FALSE if word-wrapping is not set.

See Also MLM_SETWRAP

■ MLM_RESETUNDO New

```
MLM_RESETUNDO
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

An application sends an MLM_RESETUNDO message to reset (clear) the undo flag of a multiple-line entry field (MLE). The undo flag is set whenever an operation within the MLE can be undone.

Parameters This message does not use any parameters.

Return Value The return value is TRUE if the MLE undo flag is cleared as a result of this message. Otherwise, the return value is FALSE, indicating that the undo flag was already cleared.

See Also MLM_QUERYUNDO, MLM_UNDO

■ MLM_SEARCH New

```
MLM_SEARCH
mp1 = MPFROMLONG(ulStyle);    /* search style                          */
mp2 = MPFROMP(pmlesearch);    /* address of structure with search data */
```

An application sends an MLM_SEARCH message to search for (and optionally replace) text within a multiple-line entry field (MLE).

Parameters *ulStyle* Low and high word of *mp1*. Specifies the style of the search. This parameter can be any combination of the following values:

Value	Meaning
MLFSEARCH_CASESENSITIVE	Specifies that the search is case-sensitive.
MLFSEARCH_SELECTMATCH	Specifies that if the text is found, it should be highlighted and scrolled into view (if necessary). This is identical to sending the MLM_SETSEL message.
MLFSEARCH_CHANGEALL	Specifies that all text found is to be replaced by the text in the **pchReplace** field of the **MLE_SEARCHDATA** structure.

pmlesearch Low and high word of *mp2*. Points to the **MLE_SEARCHDATA** structure that contains the search data. The **MLE_SEARCHDATA** structure has the following form:

```
typedef struct _MLE_SEARCHDATA {
   USHORT cb;
   PCHAR  pchFind;
   PCHAR  pchReplace;
   SHORT  cchFind;
   SHORT  cchReplace;
   IPT    iptStart;
   IPT    iptStop;
   USHORT cchFound;
} MLE_SEARCHDATA;
```

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

Return Value The return value is TRUE if the search is successful, or it is FALSE, indicating that the search string was not found.

Comments If the MLFSEARCH_CHANGEALL flag is not set and a match is found, the **iptStart** field of the **MLE_SEARCHDATA** structure is set to the offset (number of characters from the beginning of the text) of the first character that matches the search string. The **cchFound** field is set to the number of characters that match the search string. The current cursor position is not changed unless the MLFSEARCH_SELECTMATCH flag is set.

While the MLE is searching, it periodically sends an MLN_SEARCHPAUSE message that contains the current position of the search. You can terminate the search by returning TRUE to the MLN_SEARCHPAUSE notification message.

Example This example searches for all occurrences of the word *bonnie* and replaces it with the word *jeannette*:

```
MLE_SEARCHDATA search;
search.cb = sizeof(search);
search.pchFind = "bonnie";
search.pchReplace = "jeannette";
search.cchFind = 6;
search.cchReplace = 9;
search.iptStart = 0;    /* from the beginning of the text */
search.iptStop = -1;    /* to the end of the text         */
WinSendMsg(hwndMle, MLM_SEARCH, MLFSEARCH_CHANGEALL, (MPARAM) &search);
```

See Also MLM_SETSEL, MLN_SEARCHPAUSE, WM_CONTROL

MLM_SETBACKCOLOR New

```
MLM_SETBACKCOLOR
mp1 = MPFROMLONG((COLOR) clr);    /* color                 */
mp2 = 0L;                         /* not used, must be zero */
```

An application sends an MLM_SETBACKCOLOR message to set the background color of a multiple-line entry field (MLE).

Parameters *clr* Specifies the color. This parameter can be one of the following values:

Value	Meaning
CLR_FALSE	All color planes are zeros.
CLR_TRUE	All color planes are ones.
CLR_DEFAULT	Default value; same as CLR_NEUTRAL.
CLR_WHITE	White.
CLR_BLACK	Black.
CLR_BACKGROUND	Reset color.
CLR_BLUE	Blue.
CLR_RED	Red.
CLR_PINK	Pink.
CLR_GREEN	Green.

Value	Meaning
CLR_CYAN	Cyan.
CLR_YELLOW	Yellow.
CLR_NEUTRAL	Neutral.
CLR_DARKGRAY	Dark gray.
CLR_DARKBLUE	Dark blue.
CLR_DARKRED	Dark red.
CLR_DARKPINK	Dark pink.
CLR_DARKGREEN	Dark green.
CLR_DARKCYAN	Dark cyan.
CLR_BROWN	Brown.
CLR_PALEGRAY	Light gray.

Return Value The return value is the previous color of the background.

See Also MLM_QUERYBACKCOLOR, MLM_SETTEXTCOLOR

■ MLM_SETCHANGED New

```
MLM_SETCHANGED
mp1 = MPFROMSHORT((BOOL) fChanged);   /* changed flag          */
mp2 = 0L;                             /* not used, must be zero */
```

An application sends an MLM_SETCHANGED message to set or clear the multiple-line entry field (MLE) changed flag.

Parameters *fChanged* Low word of *mp1*. Specifies whether to set or clear the changed flag. A value of TRUE sets the changed flag.

Return Value The return value is the previous state of the MLE changed flag.

See Also MLM_QUERYCHANGED, MLN_CHANGE, WM_CONTROL

■ MLM_SETFIRSTCHAR New

```
MLM_SETFIRSTCHAR
mp1 = MPFROMLONG(lOffChar);   /* insertion point        */
mp2 = 0L;                     /* not used, must be zero */
```

An application sends an MLM_SETFIRSTCHAR message to specify the first visible character in a multiple-line entry field (MLE). The MLE scrolls the text vertically and horizontally as needed to place the character in the upper-left corner of the MLE window.

Parameters *lOffChar* Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the character to be placed in the upper-left corner of the MLE window.

Return Value The return value is always TRUE.

Comments If the value specified by the *lOffChar* parameter is greater than the total number of characters in the MLE, the first visible character is set one beyond the last character in the MLE.

See Also MLM_QUERYFIRSTCHAR

MLM_SETFONT New

```
MLM_SETFONT
mp1 = MPFROMP(pfattrs);      /* pointer to structure with font info. */
mp2 = OL;                    /* not used, must be zero              */
```

An application sends an MLM_SETFONT message to set the font for a multiple-line entry field (MLE).

Parameters *pfattrs* Low and high word of *mp1*. Points to the **FATTRS** structure that contains the font information. The **FATTRS** structure has the following form:

```
typedef struct _FATTRS {
    USHORT  usRecordLength;
    USHORT  fsSelection;
    LONG    lMatch;
    CHAR    szFacename[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lMaxBaselineExt;
    LONG    lAveCharWidth;
    USHORT  fsType;
    USHORT  fsFontUse;
} FATTRS;
```

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

Return Value The return value is TRUE if the font is successfully set or FALSE if an error occurs.

Example This example retrieves the current font information, changes it to italic, and sets it using the MLM_SETFONT message:

```
FATTRS fat;
fat.usRecordLength = sizeof(FATTRS);
WinSendMsg(hwndMle, MLM_QUERYFONT, (MPARAM) &fat, (MPARAM) OL);
fat.fsSelection = FATTR_SEL_ITALIC;
WinSendMsg(hwndMle, MLM_SETFONT, (MPARAM) &fat, (MPARAM) 0);
```

See Also MLM_QUERYFONT

MLM_SETFORMATRECT New

```
MLM_SETFORMATRECT
mp1 = MPFROMP((PMLEFORMATRECT) pmlefrmrcl); /* point to format rect. */
mp2 = MPFROMLONG((ULONG) flOptions);        /* options               */
```

An application sends an MLM_SETFORMATRECT message to set a format rectangle in a multiple-line entry field (MLE). The format rectangle can be used to limit the insertion of text within the MLE window.

Parameters

pmlefrmrcl Low and high word of *mp1*. Points to the **MLEFORMATRECT** structure that contains the format-rectangle dimensions. If this parameter is NULL, the current MLE-window dimensions (minus the border or scroll bars) is used. The **MLEFORMATRECT** structure has the following form:

```
typedef struct _MLEFORMATRECT {
    LONG cxFormat;
    LONG cyFormat;
} MLEFORMATRECT;
```

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

flOptions Low and high word of *mp2*. Specifies how the format rectangle is to be used. A value of zero causes the MLE to remove any format rectangle and to ignore the *pmlefrmrcl* parameter. Otherwise, this parameter can be a combination of the following values:

Value	Meaning
MLFFMTRECT_LIMITHORZ	Specifies that the text within the MLE cannot exceed the horizontal dimension specified by the *pmlefrmrcl* parameter. If word-wrap mode is turned on before the format rectangle is set, lines automatically wrap to stay within the horizontal limit of the format rectangle. If word-wrap mode is turned off before the format rectangle is set, an MLN_PIXHORZOVERFLOW notification message is sent to the application whenever an operation would exceed the horizontal limit specified in the format rectangle.
MLFFMTRECT_LIMITVERT	Specifies that the text within the MLE cannot exceed the vertical dimension specified by the *pmlefrmrcl* parameter. When an MLE operation would cause text to exceed the vertical limit, an MLN_PIXVERTOVERFLOW notification message is sent to the application.
MLFFMTRECT_MATCHWINDOW	Specifies that the format rectangle is to be kept the same size as the MLE window (minus the border or scroll bars).
MLFFMTRECT_FORMATRECT	Specifies that the format rectangle is to be kept the same size as the MLE window (minus the border or scroll bars) and that text cannot exceed the size of the window. This value is equivalent to combining the MLFFMTRECT_LIMITHORZ, MLFFMTRECT_LIMITVERT, and MLFFMTRECT_MATCHWINDOW values.

Return Value

The return value is TRUE if the text fits within the new format-rectangle dimensions. Otherwise, it is FALSE, indicating that the text does not fit and that the format rectangle is not set.

Comments

Whenever an insertion would cause the text to be too long for the MLE, the MLN_PIXVERTOVERFLOW or MLN_PIXHORZOVERFLOW notification message is sent.

See Also

MLM_QUERYFORMATRECT, MLN_PIXHORZOVERFLOW, MLN_PIXVERTOVERFLOW, WM_CONTROL

■ MLM_SETIMPORTEXPORT New

```
MLM_SETIMPORTEXPORT
mp1 = MPFROMP((PBYTE) pBuf);          /* pointer to buffer */
mp2 = MPFROMSHORT((USHORT) cbBuf);    /* buffer size       */
```

An application sends an MLM_SETIMPORTEXPORT message to set the transfer buffer for a multiple-line entry field (MLE).

Parameters

pBuf Low and high word of *mp1*. Points to the buffer to be used by the MLM_IMPORT, MLM_EXPORT, and MLM_SEARCH messages.

cbBuf Low word of *mp2*. Specifies the size (in bytes) of the buffer pointed to by the *pBuf* parameter. The largest size that can be specified is 65,535.

Return Value

The return value is always TRUE.

See Also

MLM_EXPORT, MLM_IMPORT

■ MLM_SETREADONLY New

```
MLM_SETREADONLY
mp1 = MPFROMSHORT(fReadOnly);    /* read-only flag         */
mp2 = 0L;                        /* not used, must be zero */
```

An application sends an MLM_SETREADONLY message to set the read-only state of a multiple-line entry field (MLE). While the read-only state is set, the user cannot change the contents of the MLE text.

Parameters

fReadOnly Low word of *mp1*. Specifies the read-only state of the MLE. A value of TRUE sets the read-only state.

Return Value

The return value is the previous value of the read-only state. If the return value is zero, the read-only state was turned off. If the return value is nonzero, the read-only state was turned on.

See Also

MLM_QUERYREADONLY

■ MLM_SETSEL New

```
MLM_SETSEL
mp1 = MPFROMLONG(lOffsetBegin);    /* offset of beginning character */
mp2 = MPFROMLONG(lOffsetEnd);      /* offset of ending character    */
```

An application sends an MLM_SETSEL message to select an area of text within a multiple-line entry field (MLE).

Parameters

lOffsetBegin Low and high word of *mp1*. Specifies the offset (number of characters from the beginning of the text) of the first character. If this parameter is set to - 1, the current cursor position is used.

lOffsetEnd Low and high word of *mp2*. Specifies the offset of the character just beyond the selection, where the cursor is to be placed. If this parameter is set to - 1, the current cursor position is used.

Return Value

The return value is always TRUE.

Comments

The MLE scrolls the text vertically and horizontally as needed to make the selection visible.

If the *lOffsetEnd* parameter is greater than the *lOffsetBegin* parameter, the cursor is placed to the right of the selected text. If *lOffsetEnd* is less than *lOffsetBegin*, the cursor is placed to the left of the selected text.

Character offsets are zero-based. Therefore, the first character has an offset of zero.

Example

This example highlights the second, third, and fourth characters of the text, and places the cursor to the right of the fourth character.

```
WinSendMsg(hwndMle, MLM_SETSEL, (MPARAM) 1L, (MPARAM) 4L);
```

See Also

MLM_QUERYSEL

■ MLM_SETTABSTOP New

```
MLM_SETTABSTOP
mp1 = MPFROMSHORT((USHORT) usTabInterval); /* tab-stop interval      */
mp2 = 0L;                                  /* not used, must be zero */
```

An application sends an MLM_SETTABSTOP message to set the interval (in pels) at which tab stops are placed in a multiple-line entry field (MLE).

Parameters

usTabInterval Low word of *mp1*. Specifies the interval (in pels) for tab stops.

Return Value

The return value is a 16-bit value (**USHORT**) that specifies the tab-stop interval.

See Also

MLM_QUERYTABSTOP

■ MLM_SETTEXTCOLOR New

```
MLM_SETTEXTCOLOR
mp1 = MPFROMLONG((COLOR) clr);    /* color                 */
mp2 = OL;                         /* not used, must be zero */
```

An application sends an MLM_SETTEXTCOLOR message to set the text color of a multiple-line entry field (MLE).

Parameters

clr Specifies the color. This parameter can be one of the following values:

Value	Meaning
CLR_FALSE	All color planes are zeros.
CLR_TRUE	All color planes are ones.
CLR_DEFAULT	Default value; same as CLR_NEUTRAL.
CLR_WHITE	White.
CLR_BLACK	Black.
CLR_BACKGROUND	Reset color.
CLR_BLUE	Blue.
CLR_RED	Red.
CLR_PINK	Pink.
CLR_GREEN	Green.
CLR_CYAN	Cyan.
CLR_YELLOW	Yellow.
CLR_NEUTRAL	Neutral.
CLR_DARKGRAY	Dark gray.
CLR_DARKBLUE	Dark blue.
CLR_DARKRED	Dark red.
CLR_DARKPINK	Dark pink.
CLR_DARKGREEN	Dark green.
CLR_DARKCYAN	Dark cyan.
CLR_BROWN	Brown.
CLR_PALEGRAY	Light gray.

Return Value The return value is the previous color of the text.

See Also MLM_QUERYTEXTCOLOR, MLM_SETBACKCOLOR

MLM_SETTEXTLIMIT New

```
MLM_SETTEXTLIMIT
mp1 = MPFROMLONG(cch);    /* maximum number of characters */
mp2 = OL;                 /* not used, must be zero       */
```

An application sends an MLM_SETTEXTLIMIT message to set the text size of a multiple-line entry field (MLE). The MLE does not accept any characters beyond this limit.

Parameters

cch Low and high word of *mp1*. Specifies the maximum number of characters allowed in the MLE. A value of - 1 specifies unlimited text is allowed.

Return Value

The return value is zero if the current MLE text is less than the new limit. Otherwise, the return value is the number of characters that exceed the specified limit, and the limit is not set.

Comments

If the user inserts more text than the specified maximum for the MLE, an MLN_TXTOVERFLOW message is sent. If the application inserts more text than the specified maximum, an MLN_OVERFLOW notification message is sent.

See Also

MLM_QUERYTEXTLIMIT, MLN_OVERFLOW, MLN_TEXTOVERFLOW, WM_CONTROL

■ MLM_SETWRAP New

```
MLM_SETWRAP
mp1 = MPFROMSHORT(fWrap);      /* word-wrap flag        */
mp2 = OL;                      /* not used, must be zero */
```

An application sends an MLM_SETWRAP message to set word-wrap mode in a multiple-line entry field (MLE).

Parameters

fWrap Low word of *mp1*. Specifies whether to turn word-wrap mode on or off. If this parameter is TRUE, word-wrapping is turned on. If it is FALSE, word-wrapping is turned off.

Return Value

The return value is TRUE if word-wrap mode is set as a result of this message. Otherwise, the return value is FALSE, indicating that the word-wrap mode cannot be changed.

Comments

Word-wrap mode affects only the visual display of the text. Line breaks inserted by the user are not affected.

Word-wrap mode cannot be turned off while the text exceeds the format rectangle specified in the MLM_SETFORMATRECT message. Word-wrap mode cannot be turned on if the result of word-wrapping would cause the text to exceed the format rectangle specified in the MLM_SETFORMATRECT message.

See Also

MLM_QUERYWRAP, MLM_SETFORMATRECT

■ MLM_UNDO New

```
MLM_UNDO
mp1 = OL;    /* not used, must be zero */
mp2 = OL;    /* not used, must be zero */
```

An application sends an MLM_UNDO message to undo a multiple-line entry-field (MLE) operation.

Parameters

This message does not use any parameters.

Return Value The return value is TRUE if an MLE operation is undone.

Comments Only the following MLE operations can be undone:

MLM_CLEAR
MLM_CUT
MLM_DELETE
MLM_INSERT
MLM_PASTE
MLM_SETBACKCOLOR
MLM_SETFONT
MLM_SETTEXTCOLOR
MLM_UNDO
WM_CHAR

If an MLM_UNDO message is sent when the undo flag has been cleared, it reverses the previous undo operation.

See Also MLM_QUERYUNDO, MLM_RESETUNDO

MLN_CHANGE — New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* MLE-window ID */
usNotifyCode = MLN_CHANGE;
```

The MLN_CHANGE notification message is sent whenever the text in a multiple-line entry field (MLE) changes.

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_CHANGE.

See Also MLM_QUERYCHANGED, MLM_SETCHANGED, WM_CONTROL

MLN_CLPBDFAIL — New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);        /* MLE-window ID */
usNotifyCode = MLN_CLPBDFAIL;
sError = (USHORT) SHORT1FROMMP(mp2);    /* error code    */
```

The MLN_CLPBDFAIL notification message is sent if the clipboard is unable to receive the text sent to it by a multiple-line entry field (MLE).

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_CLPBDFAIL.

sError Specifies the error that occurred. This parameter can be one of the following error values:

Value	Meaning
MLFCLPBD_TOOMUCHTEXT	Specifies that the amount of text exceeds the capacity of the clipboard.
MLFCLPBD_ERROR	Specifies an unknown clipboard error.

See Also MLM_COPY, MLM_CUT, WM_CONTROL

■ MLN_HSCROLL New

```
MLN_HSCROLL
id = (USHORT) SHORT1FROMMP(mp1);      /* scroll-bar window ID */
usNotifyCode = MLN_UNDOOVERFLOW;
sPos = (USHORT) SHORT1FROMMP(mp2);    /* slider position      */
```

The MLN_HSCROLL notification message is sent to the owner of the multiple-line entry field (MLE) window when a horizontal scroll event occurs.

Parameters *id* Low word of *mp1*. Identifies the scroll-bar window.

usNotifyCode High word of *mp1*. Set to MLN_HSCROLL.

sPos Low word of *mp2*. Specifies the number of pels of text (nonvisible) to the left of the window.

Return Value An application should return zero if it processes this message.

See Also MLN_VSCROLL, WM_CONTROL

■ MLN_KILLFOCUS New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* MLE-window ID */
usNotifyCode = MLN_KILLFOCUS;
```

The MLN_KILLFOCUS notification message is sent whenever the window in a multiple-line entry field (MLE) window loses the input focus.

Parameters *id* Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_KILLFOCUS.

Return Value An application should return zero if it processes this message.

See Also MLN_SETFOCUS, WM_CONTROL

MLN_MARGIN New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);            /* MLE-window ID           */
usNotifyCode = MLN_MARGIN;
pmrg = (PMARGSTRUCT) PVOIDFROMMP(mp2);      /* pointer to MLEMARGSTRUCT */
```

The MLN_MARGIN notification message is sent when the mouse moves over one of the margins of a multiple-line entry field (MLE).

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_MARGIN.

pmrg Low and high word of *mp2*. Points to the **MLEMARGSTRUCT** structure that contains the margin data. The **MLEMARGSTRUCT** structure has the following form:

```
typedef struct _MLEMARGSTRUCT {
    USHORT afMargins;
    USHORT usMouMsg;
    IPT    iptNear;
} MLEMARGSTRUCT;
```

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

Return Value An application should return zero if you want the MLE to process this message.

See Also WM_CONTROL

MLN_MEMERROR New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* MLE-window ID */
usNotifyCode = MLN_MEMERROR;
```

The MLN_MEMERROR notification message is sent if there is insufficient memory for the requested operation within a multiple-line entry field (MLE).

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_MEMERROR.

Return Value An application should return zero if it processes this message.

See Also WM_CONTROL

MLN_OVERFLOW New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);                /* MLE-window ID        */
usNotifyCode = MLN_OVERFLOW;
pmleover = (PMLEOVERFLOW) PVOIDFROMMP(mp2); /* point to MLEOVERFLOW */
```

The MLN_OVERFLOW notification message is sent when an operation in a multiple-line entry field (MLE) would overflow a text limit or a format rectangle.

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_OVERFLOW.

pmleover Low and high word of *mp2*. Points to an **MLEOVERFLOW** structure. The **MLEOVERFLOW** structure has the following form:

```
typedef struct _MLEOVERFLOW {
   ULONG afErrInd;
   LONG nBytesOver;
   LONG pixHorzOver;
   LONG pixVertOver;
}  MLEOVERFLOW;
```

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

Return Value The application should return TRUE to retry the operation.

Comments Before returning TRUE, the application should perform some operation (for example, changing the dimensions of the format rectangle) that will enable the text to fit.

Overflow caused by user-inserted text results in a MLN_PIXHORZOVERFLOW or MLN_VERTOVERFLOW notification message. Overflow caused by an application sending a message to the MLE results in a MLN_OVERFLOW message.

See Also MLN_PIXHORZOVERFLOW, MLN_PIXVERTOVERFLOW, WM_CONTROL

■ MLN_PIXHORZOVERFLOW New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);      /* MLE-window ID      */
usNotifyCode = MLN_PIXHORZOVERFLOW;
lOverFlow = LONGFROMMP(mp2);          /* amount of overflow */
```

The MLN_PIXHORZOVERFLOW notification message is sent whenever user uses the keyboard to insert more text than can fit in the current format rectangle or the text limit of a multiple-line entry field (MLE).

Parameters *id* Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_PIXHORZOVERFLOW.

lOverFlow Low and high word of *mp2*. The number of pels by which the operation overflows the current format rectangle.

Return Value An application should return TRUE to retry the operation. If the application returns FALSE, the user cannot insert additional text.

Comments Before returning TRUE, the application should perform some operation (for example, changing the dimensions of the format rectangle) that will enable the text to fit.

See Also MLN_OVERFLOW, MLN_PIXVERTOVERFLOW, WM_CONTROL

■ MLN_PIXVERTOVERFLOW New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);         /* MLE-window ID      */
usNotifyCode = MLN_PIXVERTOVERFLOW;
lOverFlow = LONGFROMMP(mp2);             /* amount of overflow */
```

The MLN_PIXVERTOVERFLOW notification message is sent whenever a user uses the keyboard to insert more text than can fit in the current format rectangle or text limit of a multiple-line entry field (MLE).

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_PIXVERTOVERFLOW.

lOverFlow Low and high word of *mp2*. The number of pels by which the operation overflowed the current format rectangle.

Return Value

An application should return TRUE to retry the operation. If the application returns FALSE, the user cannot insert additional text.

Comments

Before returning TRUE, the application should perform some operation (for example, changing the dimensions of the format rectangle) that will enable the text to fit.

Example

This example processes the MLN_PIXVERTOVERFLOW message by increasing the size of the format rectangle:

```
MLEFORMATRECT mlefr;

case MLN_PIXVERTOVERFLOW:
    mlefr.cyFormat += 100;
    WinSendMsg(hwndMle, MLM_SETFORMATRECT, (MPARAM) &mlefr,
        (MPARAM) MLFFMTRECT_LIMITVERT);
    return TRUE;
```

See Also

MLN_PIXHORZOVERFLOW, WM_CONTROL

■ MLN_SEARCHPAUSE New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);         /* MLE-window ID      */
usNotifyCode = MLN_SEARCHPAUSE;
lCurOffset = (ULONG) LONGFROMMP(mp2);    /* position of search */
```

The MLN_SEARCHPAUSE notification message is sent periodically while a multiple-line entry field (MLE) searches as a result of an MLM_SEARCH message. An application can use this message to terminate the search.

Parameters

id Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_SEARCHPAUSE.

lCurOffset Low and high word of *mp2*. Specifies the offset (number of characters from the beginning of the text) of the current character being searched for.

Return Value

The application should return FALSE to continue the search or TRUE to terminate the search.

See Also

MLM_SEARCH, WM_CONTROL

■ MLN_SETFOCUS New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);    /* MLE-window ID */
usNotifyCode = MLN_SETFOCUS;
```

The MLN_SETFOCUS notification message is sent when the window in a multiple-line entry field (MLE) receives the input focus.

Parameters *id* Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_SETFOCUS.

Return Value An application should return zero if it processes this message.

See Also MLN_KILLFOCUS, WM_CONTROL

■ MLN_TEXTOVERFLOW New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);      /* MLE-window ID          */
usNotifyCode = MLN_TEXTOVERFLOW
cchOver = (ULONG) LONGFROMMP(mp2);    /* characters over limit */
```

The MLN_TEXTOVERFLOW notification message is sent when an operation in a multiple-line entry field (MLE) exceeds the current text limit.

Parameters *id* Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_TEXTOVERFLOW.

cchOver Low and high word of *mp2*. Specifies the number of characters by which the text limit would overflow if the present operation completes.

Return Value An application should return TRUE to retry the operation. If the application returns FALSE, the user cannot insert additional text.

Comments Before returning TRUE, the application should perform some operation (for example, changing the dimensions of the format rectangle) that will enable the text to fit.

See Also MLN_OVERFLOW, WM_CONTROL

■ MLN_UNDOOVERFLOW New

```
WM_CONTROL
id = (USHORT) SHORT1FROMMP(mp1);   /* MLE-window ID */
usNotifyCode = MLN_UNDOOVERFLOW;
```

The MLN_UNDOOVERFLOW notification message is sent by a multiple-line entry field (MLE) if a text change cannot be undone because the amount of text involved exceeds the undo limit. This includes text entry, deletion, and cutting and pasting.

Parameters *id* Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_UNDOOVERFLOW.

Return Value An application should return zero if it processes this message.

See Also MLM_CUT, MLM_DELETE, MLM_INSERT, MLM_PASTE, WM_CONTROL

MLN_VSCROLL **New**

```
MLN_VSCROLL
id = (USHORT) SHORT1FROMMP(mp1);       /* control-window ID */
usNotifyCode = MLN_UNDOOVERFLOW;
sPos = (USHORT) SHORT1FROMMP(mp2);     /* slider position   */
```

The MLN_VSCROLL notification message is sent to the owner of a multiple-line entry field (MLE) window when a vertical scroll event occurs.

Parameters *id* Low word of *mp1*. Identifies the MLE window.

usNotifyCode High word of *mp1*. Set to MLN_VSCROLL.

sPos Low word of *mp2*. Specifies the top line of the display text.

Return Value An application should return zero if it processes this message.

See Also MLN_HSCROLL, WM_CONTROL

MM_DISMISSMENU **New**

```
MM_DISMISSMENU
mp1 = 0L;   /* not used, must be zero */
mp2 = 0L;   /* not used, must be zero */
```

An application sends an MM_DISMISSMENU message to dismiss a pull-down menu. Ordinarily, an application sends this message only to a pull-down menu that has the MIA_NODISMISS attribute.

Parameters This message does not use any parameters.

Return Value This message does not return a value.

See Also MM_ENDMENUMODE

MM_QUERYSELITEMID **Change**

```
MM_QUERYSELITEMID
mp1 = MPFROM2SHORT( (BOOL) fIncludeSubMenus, 0);
mp2 = 0L;    /* must be zero */
```

An application sends an MM_QUERYSELITEMID message to determine the identifier of the selected menu item.

Parameters *fIncludeSubMenus* High word of *mp1*. Specifies whether to include submenus in the search. A value of TRUE includes submenus.

Return Value The return value is the identifier of the selected item, MIT_NONE if no item is selected, or MID_ERROR if an error occurs.

See Also MM_SELECTITEM

Changes The *fIncludeSubMenus* parameter has been added.

■ MOU_DISPLAYMODECHANGE New

USHORT DosDevIOCtl(**0L, 0L, 0x005D, 0x0007,** *hDevice* **)**
HFILE *hDevice*; /* device handle */

The MOU_DISPLAYMODECHANGE function notifies the mouse device driver that a display-mode change is complete.

Parameters *hDevice* Identifies the pointing device that receives the device-control function. This handle must have been created previously 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 MOU_DISPLAYMODECHANGE function notifies the mouse that a mode switch is complete and that drawing is allowed. The pointer is redrawn if it was hidden when the mode switch began.

See Also **DosDevIOCtl, DosOpen, VioSetMode**

■ MOU_SETPROTDRAWADDRESS Change

USHORT DosDevIOCtl(*pbDrawData*, *pbFunction*, **0x005A, 0x0007,** *hDevice* **)**
PBYTE *pbDrawData*; /* pointer to drawing data */
PBYTE *pbFunction*; /* pointer to structure with drawing function */
HFILE *hDevice*; /* device handle */

The MOU_SETPROTDRAWADDRESS function notifies the mouse device driver of the address of a protected-mode pointer-draw function. This function is valid for protected mode only.

Parameters *pbDrawData* Points to the **PTRDRAWDATA** structure. This structure has the following form:

```
typedef struct _PTRDRAWDATA {
    USHORT cb;                  /* length                    */
    USHORT usConfig;            /* which display to draw on  */
    USHORT usFlag               /* Application/BVS Flag      */
} PTRDRAWDATA;
```

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

pbFunction Points to the **PTRDRAWFUNCTION** structure that contains the address of the pointer-draw function. This structure has the following form:

```
typedef struct _PTRDRAWFUNCTION {
    PFN pfnDraw;
    PCH pchDataSeg;
} PTRDRAWFUNCTION;
```

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments The pointer-draw routine is an installed, pseudo-character device driver. The mouse handler must do the following:

- Open the pointer-draw device driver.
- Query the pointer-draw device driver for the address of its entry point.
- Pass the resulting address of the pointer-draw entry point to the mouse device driver that uses this function.

See Also **DosOpen**, MOU_SETREALDRAWADDRESS

Changes The first parameter of the **DosDevIOCtl** function is now *pbDrawData*, which points to a **PTRDRAWDATA** structure.

■ MOU_SETREALDRAWADDRESS Change

USHORT DosDevIOCtl(*pvConfig*, *pbFunction*, **0x005B, 0x0007,** *hDevice* **)**
PVOID *pvConfig*; /* pointer to configuration structure */
PBYTE *pbFunction*; /* pointer to structure with function */
HFILE *hDevice*; /* device handle */

The MOU_SETREALDRAWADDRESS function notifies the real-mode mouse device driver of the entry point of a real-mode pointer-draw routine. This function is intended for use by Session Manager at the end of system initialization and is valid for real mode only.

Parameters *pvConfig* Points to the **VIOCONFIGINFO** structure that contains information about configuration of the default display. The **VIOCONFIGINFO** structure has the following format:

```
typedef struct _VIOCONFIGINFO {
    USHORT  cb      ;
    USHORT  adapter;
    USHORT  display;
    ULONG   cbMemory;
    USHORT  Configuration;
    USHORT  VDHVersion;
    USHORT  Flags;
    ULONG   HWBufferSize;
    ULONG   FullSaveSize;
    ULONG   PartSaveSize;
    USHORT  EMAdaptersOFF;
    USHORT  EMDisplaysOFF;
} VIOCONFIGINFO;
```

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

pbFunction Points to the **PTRDRAWFUNCTION** structure that contains the address of the pointer-draw function. The **PTRDRAWFUNCTION** structure has the following form:

```
typedef struct _PTRDRAWFUNCTION {
    PFN     pfnDraw;
    PCH     pchDataSeg;
} PTRDRAWFUNCTION;
```

hDevice Identifies the pointing device that receives the device-control function. The handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

See Also **DosOpen**, MOU_SETPROTDRAWADDRESS

Changes The first parameter now points to a **VIOCONFIGINFO** structure.

■ MOU_UPDATEDISPLAYMODE Change

USHORT DosDevIOCtl(*pvConfigInfo*, *pviomi*, **0x0051, 0x0007,** *hDevice* **)**

```
PVOID pvConfigInfo;        /* pointer to structure with configuration info */
PVIOMODEINFO pviomi;       /* pointer to structure with screen mode        */
HFILE hDevice;             /* device handle                                */
```

The MOU_UPDATEDISPLAYMODE function notifies the mouse device driver that the display mode has been modified.

Parameters *pvConfigInfo* Points to the **VIOCONFIGINFO** structure that contains the current display-configuration information. The **VIOCONFIGINFO** structure has the following form:

```
typedef struct _VIOCONFIGINFO {
    USHORT  cb;
    USHORT  adapter;
    USHORT  display;
    ULONG   cbMemory;
    USHORT  Configuration;
    USHORT  VDHVersion;
    USHORT  Flags;
    ULONG   HWBufferSize;
    ULONG   FullSaveSize;
    ULONG   PartSaveSize;
    USHORT  EMAdaptersOFF;
    USHORT  EMDisplaysOFF;
 } VIOCONFIGINFO;
```

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

pviomi Points to the **VIOMODEINFO** structure that contains the display-mode information. The **VIOMODEINFO** structure has the following form:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR  fmt_ID;
    UCHAR  attrib;
    ULONG  buf_addr;
    ULONG  buf_length;
    ULONG  full_length;
    ULONG  partial_length;
    PCH    ext_data_addr;
} VIOMODEINFO;
```

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

hDevice Identifies the pointing device that receives the device-control function. This handle must have been created previously by using the **DosOpen** function.

Return Value The return value is zero if the function is successful or an error value if an error occurs.

Comments When the video I/O subsystem or registered video I/O subsystem sets the display mode, it must notify the mouse device driver prior to switching display modes, in order to synchronize the mouse device driver's functions that update the pointer.

See Also **DosOpen, VioSetMode**

Changes This function has been updated to reflect changes to the **VIOMODEINFO** and **VIOCONFIGINFO** structures.

■ MOU_VER New

USHORT DosDevIOCtl(*pusVersion*, **0L, 0x006A, 0x0007,** *hDevice***)**

```
PUSHORT pusVersion;   /* pointer to version number */
HFILE hDevice;        /* device handle             */
```

The MOU_VER function returns the version number of the mouse driver.

Parameters *pusVersion* Points to a data area in which the version number of the mouse driver is returned.

hDevice Identifies the pointing device that receives the device-control function. This handle must have been created previously 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 MOU_VER function returns 0x0001 as the version number of the mouse driver to indicate that the following features are supported. These features are new for MS OS/2, version 1.2.

Function	Change
MOU_DISPLAYMODECHANGE	New **IOCtl** function.
MOU_SETPROTDRAWADDRESS	New *pbDrawData* parameter.
MOU_SETREALDRAWADDRESS	New *pvConfig* parameter.
MOU_UPDATEDISPLAYMODE	New *pvConfigInfo* parameter.
MOU_UPDATEDISPLAYMODE	Size of **VIOMODEINFO** structure increased from 12 to 34 bytes.
MOU_VER	New **IOCtl** function.

The MOU_VER function should be used to determine the version number of the mouse device driver before any of these features are used, in order to maintain compatibility with earlier versions of MS OS/2.

See Also

DosDevIOCtl, DosOpen

■ MouGetNumQueEl — Correction

USHORT MouGetNumQueEl(*pmouqi, hmou* **)**
PMOUQUEINFO *pmouqi*; /* pointer to structure for number of events */
HMOU *hmou*; /* mouse handle */

The **MouGetNumQueEl** function retrieves the number of events in the mouse-event queue.

Parameters

pmouqi Points to the **MOUQUEINFO** structure that receives the number of events in the mouse-event queue. The **MOUQUEINFO** structure has the following form:

```
typedef struct _MOUQUEINFO {
    USHORT cEvents;
    USHORT cmaxEvents;
} MOUQUEINFO;
```

hmou Identifies the mouse. This handle must have been created previously by using the **MouOpen** function.

Return Value

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

ERROR_MOUSE_NO_DEVICE

Example

This example creates a mouse handle, enables the mouse pointer to be drawn, and runs within an infinite **for** loop until there are no events in the queue:

```
HMOU hmou;
MOUEVENTINFO mouevEvent;
MOUQUEINFO mouqi;
USHORT fWait = FALSE;
MouOpen(OL, &hmou);
MouDrawPtr(hmou);
for (;;) {
    MouGetNumQueEl(&mouqi, hmou);       /* retrieves queue        */
    if (mouqi.cEvents > 1)              /* until the last queue... */
        MouReadEventQue(&mouevEvent, &fWait, hmou);
    else
        break;
}
```

See Also **MouFlushQue, MouOpen, MouReadEventQue**

Corrections The example was lacking a closing parenthesis at the end of the **MouGetNumQueEl** function call. This has been added.

■ MouSynch — Correction

USHORT MouSynch(*fWait*)
USHORT *fWait*; /* wait/no-wait flag */

The **MouSynch** function synchronizes access to the mouse. This function should be used by a **Mou** subsystem to prevent more than one process from accessing the mouse handle at any one time.

Parameters *fWait* Specifies whether to wait if the mouse device driver is currently busy. If this parameter is FALSE, the function returns control immediately without waiting for the device to become free. If this parameter is TRUE, the function waits until the mouse handle is free.

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

Comments The **MouSynch** function requests an exclusive system semaphore that clears when the **Mou** subsystem returns to the mouse router. The **MouSynch** function blocks all other threads within a screen group until the semaphore clears.

A registered mouse subsystem should not issue the **MouSynch** function when the base video subsystem (BVS) issues **MouOpen** and **MouClose** functions. A registered mouse subsystem must provide the required level of serialization for the **MouOpen** and **MouClose** functions without calling **MouSynch.** This special processing is required because **MouOpen** and **MouClose** are issued by BVS on the **VioSetMode** path. The **VioSetMode** function can be issued, in turn, by a **VioSavRedrawWait** thread. You can assume the synchronization semaphore was already held by another thread blocked by a call to the **MouReadEventQue** function.

Note that if a save/redraw wait thread issues the **VioSetMode** function, and if BVS in turn issues the **MouOpen** or **MouClose** function and the mouse subsystem in turn issues the **MouSynch** function, the screen switch will be blocked and the system will "hang."

See Also **DosCloseSem, DosDevIOCtl, MouClose MouOpen, MouReadEventQue, MouRegister, VioSavRedrawWait, VioSetMode**

Corrections

A registered mouse subsystem should not issue **MouSynch** when the base video subsystem (BVS) issues **MouOpen** and **MouClose** functions. A registered mouse subsystem must provide the required level of serialization for **MouOpen** and **MouClose** without calling **MouSynch**. This special processing is required because **MouOpen** and **MouClose** are issued by BVS on the **VioSetMode** path. The **VioSetMode** function can be issued, in turn, by a **VioSavRedrawWait** thread. You can assume the synchronization semaphore was already held by another thread blocked by a call to **MouReadEventQue**.

Note that if a save/redraw wait thread issues the **VioSetMode** function, if BVS in turn issues the **MouOpen** or **MouClose** function, and the mouse subsystem in turn issues the **MouSynch** function, the screen switch will be blocked and the system will "hang."

■ PicIchg — New

```
BOOL PicIchg( hab, pszSrcFile, pszDestFile, lType)
HAB hab;              /* anchor-block handle             */
PSZ pszSrcFile;       /* pointer to source-file name     */
PSZ pszDestFile;      /* pointer to destination-file name */
LONG lType;           /* translation type                */
```

The **PicIchg** function converts an interchange file to a metafile, or converts a symbol file to a font file.

Parameters

hab Identifies the anchor block.

pszSrcFile Points to the string that contains the name of the source file. This name must be a valid MS OS/2 filename.

pszDestFile Points to the string that contains the name of the destination file. This name must be a valid MS OS/2 filename.

lType Specifies the type of conversion requested. This parameter can be one of the following values:

Value	Meaning
PIC_PIFTOMET	Converts an interchange file to a metafile.
PIC_SSTOFONT	Converts a symbol set to a font.

Return Value

The return value is TRUE if the conversion is successful or FALSE if an error occurs.

Comments

Any reference to an internal symbol or pattern set is changed to a reference to the default font character set. Any reference to a line-type set is changed to a reference to the default line type.

Only outline fonts are supported.

■ PicPrint New

```
BOOL PicPrint( hab, pszSrcFile, lType, pszParms)
HAB hab;            /* anchor-block handle          */
PSZ pszSrcFile;     /* pointer to source-file name  */
LONG lType;         /* type of file to print        */
PSZ pszParms;       /* spooler parameters           */
```

The **PicPrint** function prints a picture file.

Parameters

hab Identifies the anchor block.

pszSrcFile Points to the string that contains the name of the source file. This name must be a valid MS OS/2 filename.

lType Specifies the type of file to print. This parameter can be one of the following values:

Value	Meaning
PIP_MF	Prints a metafile.
PIP_PIF	Prints an interchange file.

pszParms Points to the string that contains spooler parameters.

Return Value

The return value is TRUE if the print operation is successful or FALSE if an error occurs.

■ PL_ALTERED New

```
PL_ALTERED
hiniUser = HWNDFROMMP(mp1);      /* handle of user-profile file   */
hiniSystem = HWNDFROMMP(mp2);    /* handle of system-profile file */
```

A PL_ALTERED message is broadcast to all frame windows when an application calls the **PrfReset** function.

Parameters

hiniUser Low and high word of *mp1*. Identifies the user-profile file.

hiniSystem Low and high word of *mp2*. Identifies the system-profile file.

Return Value

An application should return zero if it processes this message.

See Also

PrfReset

■ PrfAddProgram New

```
HPROGRAM PrfAddProgram( hini, pprogde, hGroup)
HINI hini;                /* initialization-file handle                   */
PPROGDETAILS pprogde;     /* address of structure with program information */
HPROGRAM hGroup;          /* program-group handle                         */
```

The **PrfAddProgram** function adds a program to the program list of a group in Desktop Manager. The same program title can be used in different groups, but the program titles within a group must each be unique.

Parameters

hini Identifies the file to which the program information is added. This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

pprogde Points to the **PROGDETAILS** structure that contains program information for the program being added to Desktop Manager. The **PROGDETAILS** structure has the following form:

```
typedef struct _PROGDETAILS {
    ULONG    Length;
    PROGTYPE progt;
    USHORT   pad1[3];
    PSZ      pszTitle;
    PSZ      pszExecutable;
    PSZ      pszParameters;
    PSZ      pszStartupDir;
    PSZ      pszIcon;
    PSZ      pszEnvironment;
    SWP      swpInitial;
    USHORT   pad2[5];
} PROGDETAILS;
```

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

hGroup Identifies the program group to which the program title is added. If this parameter is zero and the *hini* parameter is HINI_USERPROFILE, the program is added to the first group defined in Desktop Manager.

Return Value

The return value is the handle for the added program if the function is successful or NULL if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_DUPLICATE_TITLE
PMERR_GROUP_PROTECTED
PMERR_INSUFF_SPACE_TO_ADD
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_PROGRAM_CATEGORY
PMERR_INVALID_TARGET_HANDLE
PMERR_INVALID_TITLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

See Also

PrfCreateGroup, PrfOpenProfile, PrfQueryDefinition, PrfQueryProgramTitles, WinAddProgram

PrfChangeProgram — New

```
BOOL PrfChangeProgram(hini, hprog, pprogde)
HINI hini;                  /* initialization-file handle                    */
HPROGRAM hprog;             /* program handle                                */
PPROGDETAILS pprogde;       /* address of structure with replacement info.   */
```

The **PrfChangeProgram** function changes the information stored in Desktop Manager about a program or group.

Parameters

hini Identifies the file that contains the program or group information to change. This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

hprog Identifies the program or group whose information is to change. If this parameter is a group handle, only the **progt** and **pszTitle** fields can be changed.

pprogde Points to the **PROGDETAILS** structure that contains the new program information. The **PROGDETAILS** structure has the following form:

```
typedef struct _PROGDETAILS {
    ULONG    Length;
    PROGTYPE progt;
    USHORT   pad1[3];
    PSZ      pszTitle;
    PSZ      pszExecutable;
    PSZ      pszParameters;
    PSZ      pszStartupDir;
    PSZ      pszIcon;
    PSZ      pszEnvironment;
    SWP      swpInitial;
    USHORT   pad2[5];
} PROGDETAILS;
```

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

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_DUPLICATE_TITLE
PMERR_GROUP_PROTECTED
PMERR_INVALID_PROGRAM_CATEGORY
PMERR_INVALID_TARGET_HANDLE
PMERR_INVALID_TITLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_IN_IDX
PMERR_UNKNOWN_APIPKT

Comments

Typically, an application calls **PrfQueryDefinition** to retrieve current information about the function, changes the returned structure, and calls **PrfChangeProgram** to change the program information.

You cannot change the program information for any program in a protected group. You can change only the visibility and the protected state.

See Also

PrfCreateGroup, PrfOpenProfile, PrfQueryDefinition

PrfCloseProfile — New

```
BOOL PrfCloseProfile(hini)
HINI hini;    /* initialization-file handle */
```

The **PrfCloseProfile** function closes a profile file opened by the **PrfOpenProfile** function.

Parameters

hini Identifies the profile file to close. The file must have been previously opened by using the **PrfOpenProfile** function.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_INVALID_INI_FILE_HANDLE

See Also

PrfOpenProfile

PrfCreateGroup — New

```
HPROGRAM PrfCreateGroup(hini, pszTitle, fsVisible)
HINI hini;          /* initialization-file handle */
PSZ pszTitle;       /* pointer to group title     */
UCHAR fsVisible;    /* visibility flag            */
```

The **PrfCreateGroup** function creates a new program-group entry in Desktop Manager. If the program group already exists, this function returns a handle to that group.

Parameters

hini Identifies the file to which the new group is added. This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

pszTitle Points to the title of the new group. The maximum string size is defined by the MAXNAMEL constant (defined in the MS OS/2 include files). Strings that exceed this limit are truncated to MAXNAMEL characters. Leading and trailing blanks are removed. The string must contain at least one nonblank character and cannot contain a backslash (\).

fsVisible Specifies the visibility of the new group. This flag can be a combination of the following values:

Value	Meaning
SHE_VISIBLE	The group is visible.
SHE_INVISIBLE	The group is invisible and cannot be viewed.
SHE_UNPROTECTED	The group is unprotected.
SHE_PROTECTED	The group is protected. Programs cannot be added or deleted from the group.

This flag can also be set or reset by using the **PrfChangeProgram** function.

Return Value The return value is the group handle for the group if the function is successful or NULL if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INSUFF_SPACE_TO_ADD
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_TARGET_HANDLE
PMERR_INVALID_TITLE
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments The new program group is empty when created. Use the **PrfAddProgram** function to add program entries to the group.

The **PrfCreateGroup** function replaces the **WinCreateGroup** function used in MS OS/2, version 1.1.

See Also **PrfAddProgram, PrfChangeProgram, WinCreateGroup**

■ PrfDestroyGroup New

```
BOOL PrfDestroyGroup(hini, hGroup)
HINI hini;                /* initialization-file handle */
HPROGRAM hGroup;          /* group handle               */
```

The **PrfDestroyGroup** function removes a group and all program information contained within that group from Desktop Manager.

Parameters *hini* Identifies the file that contains the group to remove. This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

hGroup Identifies the group to be removed from Desktop Manager.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_GROUP_PROTECTED
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_TARGET_HANDLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments You cannot remove a group that is protected. You can remove a group that contains programs.

See Also **PrfCreateGroup, PrfOpenProfile**

■ PrfOpenProfile New

```
HINI PrfOpenProfile( hab, pszProfileName)
HAB hab;                /* anchor-block handle  */
PSZ pszProfileName;     /* pointer to profile name */
```

The **PrfOpenProfile** function opens a profile file. If the profile file does not already exist, this function creates it. This function cannot be used to open the user-profile or system-profile files.

Parameters

hab Identifies the anchor block.

pszProfileName Points to the null-terminated string that contains the fully qualified filename of the profile file. If no path information is included, the default directory for the application is used. While not required, it is recommended that the extension *.ini* be used.

Return Value

The return value is a handle to the profile file if the function is successful or NULL if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_CALL_NOT_EXECUTED
PMERR_INVALID_DIRECTORY

See Also

PrfCloseProfile

■ PrfQueryDefinition New

```
ULONG PrfQueryDefinition( hini, hprog, pprogde, cbBuf)
HINI hini;                  /* initialization-file handle            */
HPROGRAM hprog;             /* program handle                        */
PPROGDETAILS pprogde;       /* address of structure for program info. */
ULONG cbBuf;                /* length of buffer for program info.    */
```

The **PrfQueryDefinition** function retrieves information about a program or program group.

Parameters

hini Identifies the file that contains the program information to retrieve. This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

hprog Identifies the program or group for which information is to be retrieved.

pprogde Points to the buffer that receives the program or group information. This buffer is formatted as a **PROGDETAILS** structure, followed by various strings pointed to by the fields within the **PROGDETAILS** structure. This buffer must be large enough for both the structure and all strings returned by this function.

The **PROGDETAILS** structure has the following form:

```
typedef struct _PROGDETAILS {
    ULONG    Length;
    PROGTYPE progt;
    USHORT   pad1[3];
    PSZ      pszTitle;
    PSZ      pszExecutable;
    PSZ      pszParameters;
    PSZ      pszStartupDir;
    PSZ      pszIcon;
    PSZ      pszEnvironment;
    SWP      swpInitial;
    USHORT   pad2[5];
} PROGDETAILS;
```

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

cbBuf Specifies the size (in bytes) of the buffer pointed to by the *pprogde* parameter. If this parameter is zero, only the length of the data is returned and the **PROGDETAILS** structure is not filled in.

Return Value

The return value is the number of bytes copied to the buffer pointed to by the *pprogde* parameter if the function is successful or zero if an error occurs. If the *cbBuf* parameter is zero, the return value is the size (in bytes) of the required buffer pointed to by the *pprogde* parameter.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_BUFFER_TOO_SMALL
PMERR_INVALID_PARM
PMERR_INVALID_PIB
PMERR_INVALID_PROGRAM_HANDLE
PMERR_INVALID_GROUP_HANDLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments

If the *hprog* parameter is a group handle, only the **progt** and **pszTitle** fields in the **PROGDETAILS** structure pointed to by *pprogde* are filled in.

The **PrfQueryDefinition** function replaces the **WinQueryDefintion** function used in MS OS/2, version 1.1.

Example

This example calls **PrfQueryDefinition** to determine the size of the buffer needed to retrieve all of the information. It then calls **DosAllocSeg** to allocate the memory and calls **PrfQueryDefinition** again to retrieve all of the program information.

```
SEL sel;
ULONG cb;
PPROGDETAILS pprogde;

/* First find the size of the buffer needed. */

cb = PrfQueryDefinition(HINI_USERPROFILE, hprog, NULL, 0L);
DosAllocSeg(cb, &sel, SEG_NONSHARED);
pprogde = MAKEP(sel, 0);
cb = PrfQueryDefinition(HINI_USERPROFILE, hprog, pprogde, cb);
```

See Also

PrfAddProgram, PrfOpenProfile, WinQueryDefinition

■ PrfQueryProfile New

BOOL PrfQueryProfile (*hab, pprfprofile* **)**
HAB *hab*; /* anchor-block handle */
PPRFPROFILE *pprfprofile*; /* address of structure for profile data */

The **PrfQueryProfile** function retrieves the fully qualified filenames of the two MS OS/2 profile (initialization) files.

Parameters

hab Identifies the anchor block.

pprfstruct Points to the **PRFPROFILE** structure that receives information about the profile filenames. The **PRFPROFILE** structure has the following form:

```
typedef struct _PRFPROFILE {
    ULONG  cchUserName;
    PSZ    pszUserName;
    ULONG  cchSysName;
    PSZ    pszSysName;
} PRFPROFILE;
```

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

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Comments

If either length field (**cchUserName** or **cchSysName**) of the **PRFPROFILE** structure is set to zero when calling this function, the length field is set to the number of bytes required to hold the corresponding filename, and that filename field is not filled in.

Example

This example calls **PrfQueryProfile** to retrieve the size of the filenames, allocates the memory needed for each string, and calls **PrfQueryProfile** again to retrieve the filenames.

```
PRFPROFILE prfpro;
SEL selUser;
SEL selSys;

prfpro.cchUserName = 0L;
prfpro.cchSysName =  0L;
PrfQueryProfile(hab, &prfpro);          /* gets size of filenames */
DosAllocSeg(prfpro.cchUserName, &selUser, SEG_NONSHARED);
DosAllocSeg(prfpro.cchSysName, &selSys, SEG_NONSHARED);
prfpro.pszUserName = MAKEP(selUser, 0);  /* initializes pointers   */
prfpro.pszSysName = MAKEP(selSys, 0);
PrfQueryProfile(hab, &prfpro);
```

See Also

PrfReset

■ PrfQueryProfileData New

```
BOOL PrfQueryProfileData( hini, pszAppName, pszKeyName, pvBuf, pcbBuf)
HINI hini;             /* initialization-file handle   */
PSZ pszAppName;        /* pointer to application name  */
PSZ pszKeyName;        /* pointer to keyname           */
PVOID pvBuf;           /* pointer to buffer            */
PULONG pcbBuf;         /* buffer length                */
```

The **PrfQueryProfileData** function retrieves binary data from the profile file. The location of the data is determined by the application name and keyname that are passed to the function.

Parameters

hini Identifies the file to query. This parameter can be a file handle obtained with **PrfOpenProfile** or one of the following values:

Value	Meaning
HINI_PROFILE	Search the user profile, and if no matching entries are found, search the system profile.
HINI_USERPROFILE	Search only the user profile.
HINI_SYSTEMPROFILE	Search only the system profile.

pszAppName Points to the null-terminated string that contains the application name. The string must be less than 1024 bytes long, including the null terminating character. The application name is case-sensitive. If *pszAppName* is NULL, a list of all application names in the profile specified by the *hini* parameter is returned.

pszKeyName Points to the null-terminated string that contains the keyname. The string must be less than 1024 bytes long, including the null terminating character. The keyname is case-sensitive. If *pszKeyName* is NULL, all keynames in the profile specified by the *hini* parameter are enumerated.

pvBuf Points to the buffer that receives the data.

pcbBuf Points to the variable that contains the size of the buffer pointed to by the *pvBuf* parameter. When the function returns, this variable contains the actual number of bytes placed in the buffer.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_PARM
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR

Comments

When a NULL is used in *pszKeyName*, if the application name specified by *pszAppName* is not found, **PrfQueryProfileData** returns FALSE.

The size of the data can be determined by calling the **PrfQueryProfileSize** function. In cases where *pvBuf* points to a list of values, the value returned by

PrfQueryProfileSize will include a NULL byte at the end of the list, used as a terminator.

See Also **PrfQueryProfileSize, PrfWriteProfileData, WinQueryProfileData**

■ PrfQueryProfileInt **New**

```
SHORT PrfQueryProfileInt( hini, pszAppName, pszKeyName, sError)
HINI hini;              /* initialization-file handle        */
PSZ pszAppName;         /* pointer to application name       */
PSZ pszKeyName;         /* pointer to keyname                */
SHORT sError;           /* value returned if keyname not found */
```

The **PrfQueryProfileInt** function retrieves an integer from the profile file.

Parameters

hini Identifies the file to query. This parameter can be a file handle obtained with **PrfOpenProfile** or one of the following values:

Value	Meaning
HINI_USERPROFILE	Search only the user profile.
HINI_SYSTEMPROFILE	Search only the system profile.

pszAppName Points to the null-terminated string that contains the application name. The string must be less than 1024 bytes long, including the null terminating character. The application name is case-sensitive.

pszKeyName Points to the null-terminated string that contains the keyname. The string must be less than 1024 bytes long, including the null terminating character. The keyname is case-sensitive.

sError Specifies the error value returned if the keyname specified by the *pszKeyName* parameter cannot be found.

Return Value

The return value is the integer representation of the text string. If the keyname cannot be found, the return value is the error value specified by the *sError* parameter.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_PARM
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_IN_IDX

Comments

The location of the integer is determined by the application name and keyname passed to this function. The **PrfWriteProfileString** function must have been used previously to store the integer as a string. For example, a string stored as "123" would be returned as the integer 123. The string may contain a leading minus sign if the number is negative.

See Also **PrfQueryProfileData, PrfWriteProfileString, WinQueryProfileInt**

■ PrfQueryProfileSize New

```
BOOL PrfQueryProfileSize( hini, pszAppName, pszKeyName, pcb)
HINI hini;           /* initialization-file handle          */
PSZ  pszAppName;     /* pointer to application name         */
PSZ  pszKeyName;     /* pointer to keyname                  */
PULONG pcb;          /* pointer to variable with data length */
```

The **PrfQueryProfileSize** function retrieves the size of the data stored at a specified location in the profile file.

Parameters

hini Identifies the file to query. This parameter can be a file handle or one of the following values:

Value	Meaning
HINI_PROFILE	Search the user profile, and if no matching entries are found, search the system profile.
HINI_USERPROFILE	Search the user profile only.
HINI_SYSTEMPROFILE	Search the system profile only.

pszAppName Points to the null-terminated string that contains the application name. The string must be less than 1024 bytes long, including the null terminating character. The application name is case-sensitive. If *pszAppName* is NULL, the length returned in the variable pointed to by the *pcb* parameter is the length required to contain a list of all application names for the *pszKeyName* parameter.

pszKeyName Points to the null-terminated string that contains the keyname. The string must be less than 1024 bytes long, including the null terminating character. The keyname is case-sensitive. If *pszKeyName* is NULL, the length returned in the variable pointed to by the *pcb* parameter is the length required to contain a list of all keynames.

pcb Points to the variable that receives the length of the data. If an error occurs, the length is not returned.

Return Value

The return value is TRUE if the function is successful.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_PARM
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR

Comments

The location of the data stored in the profile file is determined by the application name and keyname passed to this function. This function is typically called to determine how much memory to allocate before calling **PrfQueryProfileData**.

The count returned by this function will be 1 greater than that returned by **PrfQueryProfileData** or **PrfQueryProfileString** in cases where these functions will return a list. This is due to an additional NULL character used as a terminator for the entire list.

See Also

PrfQueryProfileData, PrfQueryProfileString, WinQueryProfileSize

■ PrfQueryProfileString New

```
ULONG PrfQueryProfileString( hini, pszAppName, pszKeyName, pszError, pszBuf, cchBuf)
HINI hini;               /* initialization-file handle     */
PSZ pszAppName;          /* pointer to application name    */
PSZ pszKeyName;          /* pointer to keyname             */
PSZ pszError;            /* pointer to default string      */
PSZ pszBuf;              /* pointer to buffer for string   */
ULONG cchBuf;            /* buffer size                    */
```

The **PrfQueryProfileString** function retrieves a string from the profile file. The location of the string is determined by the application name and keyname passed to this function.

Parameters

hini Identifies the file to query. This parameter can be a file handle or one of the following values:

Value	Meaning
HINI_PROFILE	Search the user profile, and if no matching entries are found, search the system profile.
HINI_USERPROFILE	Search only the user profile.
HINI_SYSTEMPROFILE	Search only the system profile.

pszAppName Points to the null-terminated string that contains the application name. The string must be less than 1024 bytes long, including the null terminating character. The application name is case-sensitive. If *pszAppName* is NULL, a list of all application names in the profile specified by the *hini* parameter is returned.

pszKeyName Points to the null-terminated string that contains the keyname. The string must be less than 1024 bytes long, including the null terminating character. The keyname is case-sensitive. If *pszKeyName* is NULL, all keynames in the profile specified by the *hini* parameter are enumerated.

pszError Points to the null-terminated string placed in the buffer pointed to by *pszBuf* if the keyname is not found.

pszBuf Points to the buffer that receives the null-terminated string.

cchBuf Specifies the length of the buffer pointed to by the *pszBuf* parameter. If the string retrieved is longer than this value, it is truncated.

Return Value

The return value is the number of characters in the buffer pointed to by *pszBuf*, or zero if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_PARM
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR

Comments

When NULL is used in *pszKeyName* and the application name specified by *pszAppName* is not found, **PrfQueryProfileString** returns FALSE.

Application data should be stored in the user profile or an application-specific profile. The system profile should be used only for system data.

See Also **PrfWriteProfileString, WinQueryProfileString**

PrfQueryProgramCategory New

```
PROGCATEGORY PrfQueryProgramCategory(hini, pszProgramName)
HINI hini;               /* initialization-file handle  */
PSZ  pszProgramName;     /* pointer to program name     */
```

The **PrfQueryProgramCategory** function retrieves the type (category) of a specified program.

Parameters

hini Identifies the file to search for program information (if the program type cannot be determined by searching the header of the executable file). This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

pszProgramName Points to the null-terminated string that contains the name of the executable file for which the type is to be returned. If the string appears to be a fully qualified path [(that is, it contains a colon (:) in the second position and/or contains a backslash (\)], the file is searched for in the indicated directory on the indicated drive. If neither of these conditions is true and the file is not in the current directory, each drive and directory specified in the path defined in the current program's environment is searched. The default extension for an executable file is *.exe*, although any extension is acceptable.

Return Value

The return value is the program category if the function is successful or zero if an error occurs. The program type can be one of the following values:

Value	Meaning
PROG_FULLSCREEN	Program runs only in a full-screen session.
PROG_WINDOWABLEVIO	Program runs in a VIO window.
PROG_PM	Program is a Presentation Manager application.
PROG_REAL	Program is a real-mode (DOS) application.
PROG_DLL	Program is a dynamic-link module.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

PMERR_DOS_ERROR

Comments

The **PrfQueryProgramCategory** function first calls the **DosQAppType** function. If the program type cannot be determined from this call, the profile specified by the *hini* parameter is searched.

Because this function calls **DosQAppType**, the program type returned may not be the same type the user specified for the program in Desktop Manager.

See Also **DosQAppType, PrfQueryDefinition**

■ PrfQueryProgramHandle New

```
ULONG PrfQueryProgramHandle(hini, pszExeName, phpga, cb, pcHandles)
HINI hini;                /* initialization-file handle             */
PSZ pszExeName;           /* pointer to executable-file name        */
PHPROGARRAY phpga;        /* address of structure for program handles */
ULONG cb;                 /* buffer size                            */
PULONG pcHandles;         /* pointer to variable for number of handles */
```

The **PrfQueryProgramHandle** function retrieves the program handles that match the name of a specified executable file.

Parameters

hini Identifies the file that contains the program information to retrieve. This parameter can be an initialization-file handle obtained by using **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

pszExeName Points to the fully qualified path [that is, it contains a colon (:) in the second position and/or contains a backslash (\)] of the executable file.

phpga Points to the **HPROGARRAY** structure that receives the program handles, one for each match found. The **HPROGARRAY** structure has the following form:

```
typedef struct _HPROGARRAY {
    HPROGRAM ahprog[1];
} HPROGARRAY;
```

cb Specifies the size (in bytes) of the buffer pointed to by the *phpga* parameter. The buffer must be large enough to hold all the program handles retrieved.

pcHandles Points to the variable that receives the number of program handles placed in the structure pointed to by the *phpga* parameter. If this value is zero when the function returns, the buffer size specified by the *cb* parameter is insufficient to hold all the program handles or an error occurred.

Return Value

The return value is the size (in bytes) of the required buffer if the function is successful. Otherwise, it is zero, indicating an error occurred or the filename was not found.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_INI_FILE_HANDLE
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_MEMORY_ALLOCATION_ERR

Comments

Typically, an application calls this function twice. The first time, the *cb* parameter is set to zero and the return value is used to determine how much memory must be allocated to hold the program handles. The second call actually retrieves the program handles.

See Also

PrfOpenProfile

PrfQueryProgramTitles New

```
ULONG PrfQueryProgramTitles(hini, hGroup, paprogti, cbBuf, pcTitles)
HINI hini;               /* initialization-file handle              */
HPROGRAM hGroup;         /* handle of group                         */
PPROGTITLE paprogti;     /* array of structures with program info.  */
ULONG cbBuf;             /* length of buffer for array of structures */
PULONG pcTitles;         /* pointer to variable for titles          */
```

The **PrfQueryProgramTitles** function retrieves information about programs within a specified group in Desktop Manager.

Parameters

hini Identifies the file that contains the program information to retrieve. This parameter can be an initialization-file handle obtained by using the **PrfOpen-Profile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

hGroup Identifies the group or program for which information is to be returned. This handle can be SGH_ROOT to retrieve information about all the groups in Desktop Manager.

paprogti Points to the buffer that receives an array of one or more **PROGTITLE** structures followed by the strings pointed to within the structures. The **PROGTITLE** structure has the following form:

```
typedef struct _PROGTITLE {
    HPROGRAM hprog;
    PROGTYPE progt;
    USHORT   pad1[3];
    PSZ      pszTitle;
} PROGTITLE;
```

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

cbBuf Specifies the total length (in bytes) of the buffer pointed to by the *paprogti* parameter.

pcTitles Points to the variable that receives the count of titles.

Return Value

The return value is the size of the required buffer if the function is successful or zero if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_BUFFER_TOO_SMALL
PMERR_INI_FILE_CORRUPT
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_PARM
PMERR_INVALID_TARGET_HANDLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NO_ENTRIES_IN_GROUP
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX
PMERR_NO_PROGRAM_FOUND

Comments

Typically an application calls this function twice. The first time, the *cbBuf* parameter is set to zero. The return value is used to allocate a sufficient buffer. Then, the application calls the function again to retrieve the program titles.

If a program handle is specified for the *hGroup* parameter, the information for only that instance of the program is returned.

See Also **PrfAddProgram, PrfOpenProfile, WinQueryProgramTitles**

PrfRemoveProgram New

```
BOOL PrfRemoveProgram( hini, hProgram )
HINI hini;                /* initialization-file handle  */
HPROGRAM hProgram;        /* program handle              */
```

The **PrfRemoveProgram** function removes a program from Desktop Manager.

Parameters

hini Identifies the file that contains the program information to remove. This parameter can be an initialization-file handle obtained by using the **PrfOpenProfile** function, or it can be the value HINI_USERPROFILE, specifying the user-profile file.

hProgram Identifies the program to remove from Desktop Manager. This parameter cannot be a group handle.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_GROUP_PROTECTED
PMERR_INVALID_INI_FILE_HANDLE
PMERR_INVALID_PIB
PMERR_INVALID_PROGRAM_HANDLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR

Comments You can remove a program from a group, even if the program is currently running. Only the program information in the group is removed—the program itself is not affected.

See Also **PrfDestroyGroup, PrfOpenProfile**

PrfReset New

```
BOOL PrfReset( hab, pprfpro )
HAB hab;             /* anchor-block handle                 */
pprfpro pprfpro;     /* address of structure with profile data */
```

The **PrfReset** function resets Presentation Manager by rereading the initialization files. This function can change which initialization files are to be used by the system.

Parameters

hab Identifies the anchor block.

pprfpro Points to the **PRFPROFILE** structure that contains the filenames of the initialization files. The **PRFPROFILE** structure has the following form:

```
typedef struct _PRFPROFILE {
    ULONG   cchUserName;
    PSZ     pszUserName;
    ULONG   cchSysName;
    PSZ     pszSysName;
} PRFPROFILE;
```

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

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_CALL_NOT_EXECUTED
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_MEMORY_SHARE
PMERR_OPEN_QUEUE
PMERR_WRITE_QUEUE

Comments The system is reset by rereading the initialization files that are specified in the **PRFPROFILE** structure. Both initialization files must be specified before calling this function.

If the path is not included as part of the initialization-file names, the current directory is used.

The **PrfReset** function modifies the **PRFPROFILE** structure passed to it. Before you can use this structure again, you must reinitialize its values.

See Also **PrfQueryProfile**

PrfWriteProfileData New

```
BOOL PrfWriteProfileData( hini, pszAppName, pszKeyName, pchBinaryData, cchData )
HINI hini;                 /* initialization-file handle    */
PSZ pszAppName;            /* pointer to application name   */
PSZ pszKeyName;            /* pointer to keyname            */
PVOID pchBinaryData;       /* pointer to data in profile file */
ULONG cchData;             /* data length                   */
```

The **PrfWriteProfileData** function places binary data in the specified profile file. The location of the data is determined by the application name and keyname passed to the function. This data can then be retrieved by using the **PrfQuery-ProfileData** function, with the application name and keyname specified in the *pszAppName* and *pszKeyName* parameters of the **PrfWriteProfileData** function.

Parameters

hini Identifies the file in which to place the binary data. This parameter can be a file handle or one of the following values:

Value	Meaning
HINI_USERPROFILE	Specifies the user profile.
HINI_SYSTEMPROFILE	Specifies the system profile.

pszAppName Points to the null-terminated string that contains the application name. The string must be less than 1024 bytes long, including the null terminating character. The application name is case-sensitive. If no application field in the profile file matches *pszAppName*, a new application field is created.

pszKeyName Points to the null-terminated string that contains the keyname. The string must be less than 1024 bytes long, including the null terminating character. If this parameter is NULL, all keynames and their data are deleted. The keyname is case-sensitive. If no keyname matches *pszKeyName*, a new keyname field is created. If the keyname already exists, the existing value is overwritten.

pchBinaryData Points to the binary data placed in the profile file. There is no explicit terminating character. If this parameter is NULL, the previous value associated with the *pszKeyName* parameter is deleted; otherwise, the data string becomes the value, even if its length is zero. The data should not exceed 64K.

cchData Specifies the size (in bytes) of the *pchBinaryData* parameter.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs. If the profile file exists but is somehow corrupted, this function returns FALSE.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_PARM
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR

Comments

The application must know the size of the stored data when it calls **PrfQueryProfileData** to retrieve the data.

See Also

PrfQueryProfileData, WinWriteProfileData

■ PrfWriteProfileString — New

```
BOOL PrfWriteProfileString( hini, pszAppName, pszKeyName, pszString)
HINI hini;            /* initialization-file handle    */
PSZ  pszAppName;      /* pointer to application name */
PSZ  pszKeyName;      /* pointer to keyname          */
PSZ  pszString;       /* pointer to string to write  */
```

The **PrfWriteProfileString** function places an ASCII string in the profile file. The location of the string is determined by the application name and keyname passed to the function. The string can then be retrieved by using the **PrfQueryProfileString** function, specifying the same application name and keyname given in the *pszAppName* and *pszKeyName* parameters of **PrfWriteProfileString**.

Parameters

hini Identifies the file to query. This parameter can be a file handle or one of the following values:

Value	Meaning
HINI_USERPROFILE	Specifies the user profile.
HINI_SYSTEMPROFILE	Specifies the system profile.

pszAppName Points to the null-terminated string that contains the application name. The string must be less than 1024 bytes long, including the null terminating character. The application name is case-sensitive. If no application field in the profile file matches *pszAppName*, a new application field is created.

pszKeyName Points to the null-terminated string that contains the keyname. The string must be less than 1024 bytes long, including the null terminating character. If *pszKeyName* is NULL, all keynames and their data are deleted. The keyname is case-sensitive. If no keyname matches *pszKeyName*, a new keyname field is created. If the keyname already exists, the existing value is overwritten.

pszString Points to the null-terminated ASCII string placed in the profile file. If *pszString* is NULL, the previous value associated with *pszKeyName* is deleted; otherwise, the ASCII string becomes the value, even if its length is zero. The string should not exceed 64K.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_PARM
PMERR_MEMORY_ALLOC
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR

Comments

User application data should be stored in either the user profile or an application specific profile. The system profile should be used only for system data, such as spooler information.

See Also

PrfQueryProfileString, WinWriteProfileString

SBM_SETTHUMBSIZE New

```
SBM_SETTHUMBSIZE
mp1 = MPFROM2SHORT((USHORT) cVisible, (USHORT) cTotal); /* items */
mp2 = 0L;                                 /* not used, must be zero */
```

An application sends an SBM_SETTHUMBSIZE message to set the size of the slider in the scroll bar.

Parameters

cVisible Low word of *mp1*. Specifies the number of visible items.

cTotal High word of *mp1*. Specifies the total number of items.

Return Value

The return value is always TRUE.

Comments

The SBM_SETTHUMBSIZE message is usually sent when the scroll bar is initialized or when the client window changes size. MS OS/2 uses the two parameters to calculate the percentage of data visible and thus the percentage of the scroll bar that the slider should occupy.

See Also

SBM_QUERYPOS, SBM_QUERYRANGE, SBM_SETPOS

■ SCR_ALLOCLDT — New

```
USHORT DosDevIOCtl(psel, pvAddrInfo, 0x0070, 0x0003, hDevice)
PSEL psel;              /* pointer to LDT selector                */
PVOID pvAddrInfo;       /* pointer to structure with address info */
HFILE hDevice;          /* device handle                          */
```

The SCR_ALLOCLDT function allocates a logical descriptor table (LDT) selector for an area of memory.

Parameters

psel Points to the logical descriptor table selector for the memory area specified by the **LDTADDRINFO** structure.

pvAddrInfo Points to the **LDTADDRINFO** structure that contains the address and size of memory for which a selector is requested.

The **LDTADDRINFO** structure has the following form:

```
typedef struct _LDTADDRINFO {
    PULONG  pulPhysAddr;
    USHORT  cb;
} LDTADDRINFO;
```

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

hDevice Identifies the screen device that receives the device-control function. This handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or the error value ERROR_I24_INVALID_PARAMETER if an error occurs.

Comments

Read/Write access is granted to data areas completely contained in the address range 0xA0000 through 0xBFFFF. Read-only access is granted to data areas outside this range, but inside the range 0x00000 through 0xFFFFF. Attempts to access any address outside this range results in an error.

See Also

SCR_ALLOCLDTOFF, SCR_DEALLOCLDT

■ SCR_ALLOCLDTOFF — New

```
USHORT DosDevIOCtl(ppv, pvAddrInfo, 0x0075, 0x0003, hDevice)
PVOID FAR * ppv;        /* pointer to variable to receive selector:offset */
PVOID pvAddrInfo;       /* pointer to structure with address info         */
HFILE hDevice;          /* device handle                                  */
```

The SCR_ALLOCLDTOFF function allocates a logical descriptor table (LDT) selector and offset for an area of memory.

Parameters

ppv Points to the variable that receives the allocated selector and offset.

pvAddrInfo Points to the **LDTADDRINFO** structure that contains the address and size of memory for which a selector is requested.

The **LDTADDRINFO** structure has the following form:

```
typedef struct _LDTADDRINFO {
    PULONG  pulPhysAddr;
    USHORT  cb;
} LDTADDRINFO;
```

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

hDevice Identifies the screen device that receives the device-control function. This handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or the error ERROR_I24_INVALID_PARAMETER if an error occurs.

Comments

Read/Write access is granted to data areas completely contained in the address range 0xA0000 through 0xBFFFF. Read-only access is granted to data areas outside this range, but inside the range 0x00000 through 0xFFFFF. Attempts to access any address outside this range results in an error.

See Also

SCR_ALLOCLDT, SCR_DEALLOCLDT

■ SCR_DEALLOCLDT New

USHORT DosDevIOCtl(0L, ***psel*****, 0x0071, 0x0003,** ***hDevice*****)**
PSEL ***psel*****;** /* pointer to LDT selector */
HFILE ***hDevice*****;** /* device handle */

The SCR_DEALLOCLDT function deallocates a logical descriptor table (LDT) selector previously allocated by the SCR_ALLOCLDT or SCR_ALLOCLDTOFF function.

Parameters

psel Points to the logical descriptor table selector to be deallocated.

hDevice Identifies the screen device that receives the device-control function. This handle must have been created previously by using the **DosOpen** function.

Return Value

The return value is zero if the function is successful or the error value ERROR_I24_INVALID_PARAMETER if an error occurs.

See Also

SCR_ALLOCLDT, SCR_ALLOCLDTOFF

■ TBM_TRACKMOVE New

```
TBM_TRACKMOVE
mp1 = MPFROMSHORT(fs);       /* tracking options       */
mp2 = 0L;                    /* not used, must be zero */
```

An application sends a TBM_TRACKMOVE message to a title-bar window control to move its owner window.

A WM_QUERYTRACKINFO message is first sent to the owner of the title-bar window control. If the return value is TRUE, the window is moved; otherwise, the operation terminates.

Parameters

fs Low word of *mp1*. Specifies tracking options. This parameter can be a combination of the following values:

Option	Meaning
TF_LEFT	Tracks the left side of the rectangle.
TF_TOP	Tracks the top of the rectangle.
TF_RIGHT	Tracks the right of the rectangle.
TF_BOTTOM	Tracks the bottom of the rectangle.
TF_MOVE	Tracks all sides of the rectangle.
TF_POINTERPOS	Repositions the pointer according to the other options specified.
TF_LEFT	Vertically centers the pointer at the left of the tracking rectangle.
TF_TOP	Horizontally centers the pointer at the top of the tracking rectangle.
TF_RIGHT	Vertically centers the pointer at the right of the tracking rectangle.
TF_BOTTOM	Horizontally centers the pointer at the bottom of the tracking rectangle.
TF_MOVE	Centers the pointer in the tracking rectangle.
TF_GRID	Restricts tracking to a predetermined grid.
TF_STANDARD	The width, height, grid width, and grid height are all multiples of the border width and border height.
TF_ALINBOUNDARY	Tracks so that no part of the tracking rectangle ever falls outside the bounding rectangle.
TF_PARTINBOUNDARY	Tracks so that the corresponding edge of the tracking rectangle is kept within the opposite edge of the boundary rectangle.

Return Value

The return value is TRUE if the operation is successful or FALSE if an error occurs.

See Also

WM_QUERYTRACKINFO

■ VioCreatePS — Correction

```
USHORT VioCreatePS( phvps, cRows, cColumns, fFormat, cAttrBytes, hvps)
PHVPS phvps;        /* pointer to variable for presentation-space handle */
SHORT cRows;        /* height of presentation space                      */
SHORT cColumns;     /* width of presentation space                       */
SHORT fFormat;      /* format of attribute byte(s)                       */
SHORT cAttrBytes;   /* number of attributes                              */
HVPS hvps;          /* presentation-space handle                         */
```

The **VioCreatePS** function creates an advanced video-input-and-output (AVIO) presentation space, the size of which must not exceed 64K. To determine the size of the presentation space, multiply the *cColumns*, *cRows*, and *cAttrBytes* parameters as follows: *cColumns* × *cRows* × (*cAttrBytes* + 1).

Parameters

phvps Points to the variable that receives the presentation-space handle. You may use this handle in subsequent **Vio** functions.

cRows Specifies the height (in character cells) of the presentation space.

cColumns Specifies the width (in character cells) of the presentation space.

fFormat Identifies the format of the attribute byte(s) in the presentation space. The content of the attribute bytes depends on the format. Currently, the only defined format is zero. If the format is zero, the attribute bytes have the following meanings:

Value	Meaning
FORMAT_CGA	Specifies a CGA format of two attribute bytes. The first byte contains the character value. The second byte contains bit fields that specify the background and foreground colors. Blink and intensity fields are not supported.
FORMAT_4BYTE	Specifies an extended format of four attribute bytes. The first byte contains the character value. The second byte contains bit fields that specify the background and foreground colors. The third byte contains bit fields that specify the underscore, reverse video, the background opacity, and the font identifier. The fourth byte is an extra byte to be used by programs.

cAttrBytes Specifies the number of attribute bytes per character cell in the presentation space. This number may be 1 or 3.

hvps Identifies the AVIO presentation space. This parameter must be zero.

Return Value

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

See Also

VioDestroyPS

Corrections

The presentation space must not exceed 64K, not 32K as previously documented.

■ VioGetBuf — Correction

USHORT VioGetBuf(*pulLVB*, *pcbLVB*, *hvio* **)**

```
PULONG pulLVB;      /* pointer to variable for address of LVB */
PUSHORT pcbLVB;     /* pointer to variable for length of LVB  */
HVIO hvio;          /* video handle                           */
```

The **VioGetBuf** function retrieves the address of the logical video buffer (LVB) that contains the current character attributes for the text output of a process. The logical video buffer is available for text-mode screens only.

A process can access and modify the contents of the logical video buffer at any time, even if the process is in the background. Changes made to the logical video buffer do not affect the physical screen until the process calls the **VioShowBuf** function.

Parameters

pulLVB Points to the variable that receives the address of the logical video buffer.

pcbLVB Points to the variable that specifies the length (in bytes) of the logical video buffer. You can use the **VioGetMode** function to determine the dimensions of the buffer.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE

Comments

If the process calling **VioGetBuf** is in the foreground, all VIO output calls are written to both the physical display buffer and the logical video buffer.

If the **VioSetMode** function is called following a call to **VioGetBuf**, the size of the logical video buffer is adjusted to correspond to the new mode.

There is one logical video buffer per session (or presentation space, for an AVIO application).

Example

This example calls **VioGetBuf** to retrieve the address of the logical video buffer. It sets the character attributes in the buffer for foreground blinking by using the OR operator to set the high bit, then it calls the **VioShowBuf** function to display the character attributes:

```
PBYTE pbLVB;
USHORT cbLVB, i;
VioGetBuf((PULONG) &pbLVB, &cbLVB, 0);
for (i = 0; i < cbLVB; i += 2)

    /* OR in the high bit to make it a blinking attribute */

    *(pbLVB + i + 1) = *(pbLVB + i + 1) | 0x80;
VioShowBuf(0, cbLVB, 0);            /* displays buffer      */
```

See Also

VioGetMode, VioGetPhysBuf, VioShowBuf

Corrections

This function is not a Family API function.

The physical and logical video buffers are not always identical.

■ VioGetConfig — Change

```
USHORT VioGetConfig( usConfigId, pvioin, hvio)
USHORT usConfigId;          /* configuration ID                    */
PVIOCONFIGINFO pvioin;      /* pointer to structure for configuration */
HVIO hvio;                  /* video handle                        */
```

The **VioGetConfig** function retrieves the video-display configuration, which defines the type of display adapter, the type of display, and the amount of video memory available in the current, primary, or secondary display.

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

Parameters

usConfigId Specifies the display adapter to retrieve the configuration for. This parameter can be one of the following values:

Value	Meaning
VIO_CONFIG_CURRENT	The current display adapter
VIO_CONFIG_PRIMARY	The primary display adapter
VIO_CONFIG_SECONDARY	The secondary display adapter

pvioin Points to the **VIOCONFIGINFO** structure that receives the display configuration for the primary display adapter. The **VIOCONFIGINFO** structure has the following form:

```
typedef struct _VIOCONFIGINFO {
    USHORT cb;
    USHORT adapter;
    USHORT display;
    ULONG cbMemory;
    USHORT config;
    USHORT dd_ver;
    USHORT flags;
    ULONG hwbuf;
    ULONG maxfullbuf;
    ULONG maxpartbuf;
    USHORT adaptptr;
    USHORT dispptr;
    USHORT cwadapt;
    USHORT adaptdata[1];
    USHORT cwdisp;
    USHORT dispdata[1];
} VIOCONFIGINFO;
```

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

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_INVALID_LENGTH
ERROR_VIO_INVALID_PARMS

Comments

MS OS/2 derives the values for the **adapter** and **display** fields of the **VIOCONFIGINFO** structure for the display configuration by using various tests, including checking the switch settings on the card.

Example

This example calls **VioGetConfig** to determine whether the primary display type is an enhanced color display:

```
VIOCONFIGINFO vioinConfig;
vioinConfig.cb = sizeof(vioinConfig);     /* structure length   */
VioGetConfig(VIO_CONFIG_PRIMARY,
    &vioinConfig,                         /* configuration data */
    0);                                   /* video handle       */
if (vioinConfig.display == DISPLAY_EGA)
    VioWrtTTY("Enhanced color display0, 24, 0);
```

See Also

VioGetMode, VioGetState

Changes

The first parameter changed from *usReserved* to *usConfigId*, allowing you to specify which display adapter to get the configuration information from.

The **VIOCONFIGINFO** structure pointed to by the *pvioin* parameter contains additional fields when used in MS OS/2, version 1.2.

■ VioGetMode — Change

USHORT VioGetMode(*pviomi, hvio* **)**

```
PVIOMODEINFO pviomi;   /* pointer to structure for screen-mode information */
HVIO hvio;             /* video handle                                     */
```

The **VioGetMode** function retrieves the current screen mode. The screen mode defines the display mode (text or graphics), the number of colors being used (2, 4, or 16), and the width and height of the screen in both character cells and pels.

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

Parameters

pviomi Points to the **VIOMODEINFO** structure that receives the screen-mode information. The **VIOMODEINFO** structure has the following form:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR attribfmt;
    UCHAR attribcount;
    ULONG pdbaddr;
    ULONG pdblen;
    ULONG fullbufsz;
    ULONG partbufsz;
    ULONG edaaddr;
} VIOMODEINFO;
```

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

hvio This parameter must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH

Comments The *hvio* parameter can be only NULL. This function cannot be used by an advanced video-input-and-output application.

Example This example calls **VioGetMode** to retrieve the mode information for the screen:

```
VIOMODEINFO viomi;
viomi.cb = sizeof(viomi);
VioGetMode(&viomi, 0);
if (viomi.fbType == 0)
    VioWrtTTY("Monochrome display\n\r", 20, 0);
```

See Also **VioCreatePS, VioGetState, VioSetMode**

Changes The **VIOMODEINFO** structure pointed to by the *pviomi* parameter contains several additional fields when used in MS OS/2,version 1.2.

Corrections The *hvio* parameter can be only NULL. This function cannot be used by an advanced video-input-and-output application.

■ VioGetState Change

USHORT VioGetState (*pvoidState, hvio* **)**
PVOID *pvoidState*; /* pointer to structure for state information */
HVIO *hvio*; /* video handle */

The **VioGetState** function retrieves the current settings of the screen-palette registers, the overscan (border) color, the blink/background intensity switch, the screen color, the underline position, or the target display.

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

Parameters *pvoidState* Points to the structure that receives the state information. The structure type, which depends on the request type specified in the **type** field of each structure, is one of the following: **VIOPALSTATE, VIOOVERSCAN, VIOINTENSITY, VIOCOLORREG, VIOSETULINELOC**, or **VIOSETTARGET**. These structures have the following forms:

```
typedef struct _VIOPALSTATE {
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;

typedef struct _VIOOVERSCAN {
    USHORT cb;
    USHORT type;
    USHORT color;
} VIOOVERSCAN;

typedef struct _VIOINTENSITY {
    USHORT cb;
    USHORT type;
    USHORT fs;
} VIOINTENSITY;

typedef struct _VIOCOLORREG {
    USHORT  cb;
    USHORT  type;
    USHORT  firstcolorreg;
    USHORT  numcolorregs;
    PCH     colorregaddr;
} VIOCOLOR;
```

```
typedef struct _VIOSETULINELOC {
    USHORT  cb;
    USHORT  type;
    USHORT  scanline;
} VIOUNDERLINE;

typedef struct _VIOSETTARGET {
    USHORT  cb;
    USHORT  type;
    USHORT  defaultalgorithm;
} VIOTARGET;
```

For each structure, you must set the **cb** and **type** fields before calling the function. Not all values for the **type** field are valid for all screen modes.

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

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created by using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH

Example

This example calls the **VioGetState** function to retrieve the settings for each of the 16 palette registers:

```
BYTE abState[38];
PVIOPALSTATE pviopal;
pviopal = (PVIOPALSTATE) abState;
pviopal->cb = sizeof(abState); /* structure size              */
pviopal->type = 0;             /* retrieves palette registers */
pviopal->iFirst = 0;           /* first palette register to return */
VioGetState(pviopal, 0);
```

See Also

VioCreatePS, VioGetMode, VioSetState

Changes

The **VIOCOLORREG, VIOSETULINELOC,** and **VIOSETTARGET** structures have been added to the list of possible structures for this function.

Corrections

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

■ VioReadCellStr — Correction

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 combinations) from the screen, starting at the specified location. If the string is longer than the current line, the function continues reading at the beginning of the next line but does not read past the end of the screen.

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

Parameters

pchCellString Points to the buffer that receives the cell string.

pcb Points to the variable that specifies the length (in bytes) of the buffer pointed to by *pchCellString*. The length should be an even number. On return, this function copies the length of the string to the variable.

usRow Specifies the row at which to begin reading the cell string.

usColumn Specifies the column at which to begin reading the cell string.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls **VioReadCellStr** to read Line 0, then calls the **VioWrtCellStr** function to write the cell string to Line 24:

```
CHAR achCells[160];
USHORT cb = sizeof(achCells);
VioReadCellStr(achCells,    /* buffer for string                  */
    &cb,                    /* points 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

Corrections

The references to cells have been changed to reflect that an attribute can be longer than one byte.

■ VioScrollDn — Correction

```
USHORT VioScrollDn(usTopRow, usLeftCol, usBotRow, usRightCol, cbLines, pbCell, hvio)
USHORT usTopRow;     /* top row                  */
USHORT usLeftCol;    /* left column              */
USHORT usBotRow;     /* bottom row               */
USHORT usRightCol;   /* right column             */
USHORT cbLines;      /* number of blank lines    */
PBYTE pbCell;        /* pointer to cell to write */
HVIO hvio;           /* video handle             */
```

The **VioScrollDn** function scrolls the current screen downward.

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

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbLines Specifies the number of lines to be inserted at the top of the screen area being scrolled. If this parameter is zero, no lines are scrolled.

pbCell Points to a character/attribute combination, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbLines*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollDn** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbLines* to their maximum values. The function clears the screen by using the character/attribute combination pointed to by the *pbCell* parameter.

Example

This example creates a cell containing the space character (0x20) and a white character attribute (0x07 on an EGA color monitor), and calls **VioScrollDn** to clear the screen by using this cell. By changing the character attribute, you could change the background color of the screen while clearing it at the same time (using the value 0xFFFF for *usBotRow*, *usRightCol*, and *cbLines* clears the screen):

```
BYTE bCell[2];
bCell[0] = 0x20;        /* space character         */
bCell[1] = 0x07;        /* white attribute (EGA)   */
VioScrollDn(0,          /* top row                 */
    0,                  /* left column             */
    0xFFFF,             /* bottom row              */
    0xFFFF,             /* right column            */
    0xFFFF,             /* number of lines         */
    bCell,              /* cell to write           */
    0);                 /* video handle            */
```

See Also

VioCreatePS, VioScrollLf, VioScrollRt, VioScrollUp

Corrections

The references to cells have been changed to reflect that an attribute can be longer than one byte.

■ VioScrollLf **Correction**

```
USHORT VioScrollLf( usTopRow, usLeftCol, usBotRow, usRightCol, cbColumns, pbCell, hvio )
USHORT usTopRow;     /* top row                      */
USHORT usLeftCol;    /* left column                  */
USHORT usBotRow;     /* bottom row                   */
USHORT usRightCol;   /* right column                 */
USHORT cbColumns;    /* number of blank columns      */
PBYTE pbCell;        /* pointer to the cell to write */
HVIO hvio;           /* video handle                 */
```

The **VioScrollLf** function scrolls the current screen toward the left.

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

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbColumns Specifies the number of columns of spaces to be inserted at the right. If this parameter is zero, no columns are inserted.

pbCell Points to a character/attribute combination, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbColumns*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollLf** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbColumns* to their maximum values. The function clears the screen by using the character/attribute combination pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollLf** to fill the last ten columns at the right of the screen with red hearts on a black background (a value of 0xFFFF is used for *usBotRow* and *usRightCol*):

```
BYTE bCell[2];
bCell[0] = 0x03;        /* heart character       */
bCell[1] = 0x04;        /* red attribute (EGA)  */
VioScrollLf(0,          /* top row               */
    0,                  /* left column           */
    0xFFFF,             /* bottom row            */
    0xFFFF,             /* right column          */
    10,                 /* columns               */
    bCell,              /* cell to write         */
    0);                 /* video handle          */
```

See Also **VioCreatePS, VioScrollDn, VioScrollRt, VioScrollUp**

Corrections The references to cells have been changed to reflect that an attribute can be longer than one byte.

■ VioScrollRt — Correction

```
USHORT VioScrollRt(usTopRow, usLeftCol, usBotRow, usRightCol, cbColumns, pbCell, hvio)
USHORT usTopRow;     /* top row                    */
USHORT usLeftCol;    /* left column                */
USHORT usBotRow;     /* bottom row                 */
USHORT usRightCol;   /* right column               */
USHORT cbColumns;    /* number of blank columns    */
PBYTE pbCell;        /* pointer to cell to write   */
HVIO hvio;           /* video handle               */
```

The **VioScrollRt** function scrolls the current screen toward the right.

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

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbColumns Specifies the number of columns of spaces to be inserted at the left. If this parameter is zero, no columns are inserted.

pbCell Points to a character/attribute combination, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow, usLeftCol, usBotRow, usRightCol,* or *cbColumns,* the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollUp** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbColumns* to their maximum values. The function clears the screen by using the character/attribute combination pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollRt** to fill the first ten columns at the left of the screen with red hearts on a black background (a value of 0xFFFF is used for *usBotRow* and *usRightCol*):

```
BYTE bCell[2];
bCell[0] = 0x03;        /* heart character       */
bCell[1] = 0x04;        /* red attribute (EGA)   */
VioScrollRt(0,          /* top row               */
    0,                  /* left column           */
    0xFFFF,             /* bottom row            */
    0xFFFF,             /* right column          */
    10,                 /* columns               */
    bCell,              /* cell to write         */
    0);                 /* video handle          */
```

See Also

VioCreatePS, VioScrollDn, VioScrollLf, VioScrollUp

Corrections

The references to cells have been changed to reflect that an attribute can be longer than one byte.

■ VioScrollUp — Correction

USHORT VioScrollUp(*usTopRow, usLeftCol, usBotRow, usRightCol, cbLines, pbCell, hvio* **)**

```
USHORT usTopRow;     /* top row                   */
USHORT usLeftCol;    /* left column               */
USHORT usBotRow;     /* bottom row                */
USHORT usRightCol;   /* right column              */
USHORT cbLines;      /* number of blank lines     */
PBYTE pbCell;        /* pointer to cell to write  */
HVIO hvio;           /* video handle              */
```

The **VioScrollUp** function scrolls the current screen upward.

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

Parameters

usTopRow Specifies the top row of the screen area to scroll.

usLeftCol Specifies the leftmost column of the screen area to scroll.

usBotRow Specifies the bottom row of the screen area to scroll.

usRightCol Specifies the rightmost column of the screen area to scroll.

cbLines Specifies the number of blank lines to insert at the bottom of the screen area being scrolled. If this parameter is zero, no lines are inserted.

pbCell Points to a character/attribute combination, called a cell, that fills the screen area left blank by the scrolling.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Comments

If the *usTopRow* and *usLeftCol* parameters are zero, they identify the upper-left corner of the screen. If you specify a value greater than the maximum for *usTopRow*, *usLeftCol*, *usBotRow*, *usRightCol*, or *cbLines*, the maximum value for that parameter is used. Maximum values depend upon the dimensions of the screen being used.

You can use the **VioScrollUp** function to clear the screen by setting *usTopRow* and *usLeftCol* to zero and *usBotRow*, *usRightCol*, and *cbLines* to their maximum values. The function clears the screen by using the character/attribute combination pointed to by the *pbCell* parameter.

Example

This example calls **VioScrollUp** to scroll the entire screen up (by using the value 0xFFFF for *usBotRow*, *usRightCol*, and *cbLines*) and to fill the screen area left blank by the scrolling with spaces on a green background (0x22 on an EGA color monitor):

```
BYTE bCell[2];
bCell[0] = 0x20;          /* space character         */
bCell[1] = 0x22;          /* green attribute (EGA)   */
VioScrollUp(0,            /* top row                 */
    0,                    /* left column             */
    0xFFFF,               /* bottom row              */
    0xFFFF,               /* right column            */
    0xFFFF,               /* number of lines         */
    bCell,                /* cell to write           */
    0);                   /* video handle            */
VioSetCurPos(0, 0, 0);
```

See Also

VioCreatePS, VioScrollDn, VioScrollLf, VioScrollRt

Corrections

The references to cells have been changed to reflect that an attribute can be longer than one byte.

■ VioSetCurType — Change

```
USHORT VioSetCurType(pvioci, hvio)
PVIOCURSORINFO pvioci;   /* pointer to structure for cursor characteristics */
HVIO hvio;               /* video handle                                    */
```

The **VioSetCurType** function sets the cursor type. The cursor is a shared resource for all processes in a screen group. If one process changes it, it is changed for all processes in the group.

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

Parameters

pvioci Points to the **VIOCURSORINFO** structure that specifies the characteristics of the cursor. The **VIOCURSORINFO** structure has the following form:

```
typedef struct _VIOCURSORINFO {
    USHORT yStart;
    USHORT cEnd;
    USHORT cx;
    USHORT attr;
} VIOCURSORINFO;
```

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously by using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_WIDTH

Comments

The **yStart** and **cEnd** fields of the **VIOCURSORINFO** structure can be set to values that are independent of the number of scan lines in the character cell. If you specify percentages for these values, MS OS/2 calculates the beginning and ending scan lines by multiplying the specified percentage by the number of scan lines in the character cell and rounding the total to the nearest scan line. Percentages are specified as a number in the range 0 through -100. For example, if **yStart** is set to -90 and **cEnd** is set to -100, the cursor occupies the bottom 10 percent of the character cell.

Example

This example calls the **VioSetCurType** function to set the current cursor type to a block cursor with 14 scan lines:

```
VIOCURSORINFO vioci;
vioci.yStart = 0;       /* beginning scan line for cursor */
vioci.cEnd = 13;        /* ending scan line, zero-based   */
vioci.cx = 0;           /* default width, one character   */
vioci.attr = 0;         /* normal attribute               */
VioSetCurType(&vioci, 0);
```

See Also

VioCreatePS, VioGetCurType, VioSetCurPos

Changes

The **yStart** and **cEnd** fields of the **VIOCURSORINFO** structure can be set to values that are independent of the number of scan lines in the character cell. If you specify percentages for these values, MS OS/2 calculates the beginning and ending scan lines by multiplying the specified percentage by the number of scan lines in the character cell and rounding the total to the nearest scan line. Percentages are specified as a number in the range 0 through -100. For example, if **yStart** is set to -90 and **cEnd** is set to -100, the cursor occupies the bottom 10 percent of the character cell.

■ VioSetMode — Change

USHORT VioSetMode (*pviomi, hvio* **)**
PVIOMODEINFO *pviomi*; /* pointer to structure for screen mode */
HVIO *hvio*; /* video handle */

The **VioSetMode** function sets the screen mode. The screen mode defines the display mode (text or graphics), the number of colors being used (2, 4, or 16), and the width and height of the screen in both character cells and pels. **VioSetMode** also initializes the cursor position and type, but does not clear the screen.

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

Parameters

pviomi Points to the **VIOMODEINFO** structure that specifies the screen mode. The **VIOMODEINFO** structure has the following form:

```
typedef struct _VIOMODEINFO {
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR attribfmt;
    UCHAR attribcount;
    ULONG pdbaddr;
    ULONG pdblen;
    ULONG fullbufsz;
    ULONG partbufsz;
    ULONG edaaddr;
} VIOMODEINFO;
```

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

hvio This parameter must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH
ERROR_VIO_MODE

Comments

Not all screen-mode values are valid for all displays.

The *hvio* parameter can be only NULL. This function cannot be used by an advanced video-input-and-output application.

When **VioSetMode** is called from a VIO-window application (as opposed to an application that is running in its own screen group), it does not change the size of a character cell.

Example

This example calls the **VioGetMode** function to retrieve the current display mode, changes the mode, and calls **VioSetMode** to enable the new display mode.

```
VIOMODEINFO viomi;
viomi.cb = sizeof(viomi);
VioGetMode(&viomi, 0);
if (viomi.vres > 350)                                 /* VGA display */
    viomi.row = (viomi.row == 50) ? 25 : 50;
else                                                  /* EGA display */
    viomi.row = (viomi.row == 43) ? 25 : 43;
VioSetMode(&viomi, 0);
```

See Also

VioCreatePS, VioGetMode, VioSetState

Changes

The **VIOMODEINFO** structure pointed to by the *pviomi* parameter contains several additional fields when used in MS OS/2, version 1.2.

Corrections

The *hvio* parameter can be only NULL. This function cannot be used by an advanced video-input-and-output application.

■ VioSetState — Change

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, the blink/background intensity, the screen color, the underline position, or the display adapter.

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

Parameters

pvoidState Points to the structure that contains the request type and the values to set. The structure type, which depends on the request type specified in the **type** field of each structure, is one of the following: **VIOPALSTATE, VIOOVERSCAN, VIOINTENSITY, VIOCOLORREG, VIOSETULINELOC**, or **VIOSETTARGET**. These structures have the following forms:

```
typedef struct _VIOPALSTATE {
    USHORT cb;
    USHORT type;
    USHORT iFirst;
    USHORT acolor[1];
} VIOPALSTATE;

typedef struct _VIOOVERSCAN {
    USHORT cb;
    USHORT type;
    USHORT color;
} VIOOVERSCAN;

typedef struct _VIOINTENSITY {
    USHORT cb;
    USHORT type;
    USHORT fs;
} VIOINTENSITY;

typedef struct _VIOCOLORREG {
    USHORT  cb;
    USHORT  type;
    USHORT  firstcolorreg;
    USHORT  numcolorregs;
    PCH     colorregaddr;
} VIOCOLOR;

typedef struct _VIOSETULINELOC {
    USHORT  cb;
    USHORT  type;
    USHORT  scanline;
} VIOUNDERLINE;

typedef struct _VIOSETTARGET {
    USHORT  cb;
    USHORT  type;
    USHORT  defaultalgorithm;
} VIOTARGET;
```

Not all request-type values are valid for all screen modes.

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

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created by using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_INVALID_LENGTH

Example

This example retrieves the current settings of the palette registers, switches palette registers #0 and #7, and calls **VioSetState** to enable the new settings:

```
BYTE abState[38];
PVIOPALSTATE pviopal;
USHORT usTmp;
pviopal = (PVIOPALSTATE) abState;
pviopal->cb = sizeof(abState);
pviopal->type = 0;              /* retrieves palette registers */
pviopal->iFirst = 0;            /* first register to retrieve  */
VioGetState(pviopal, 0);        /* retrieves current settings  */
usTmp = pviopal->acolor[0];     /* swaps #0 and #7             */
pviopal->acolor[0] = pviopal->acolor[7];
pviopal->acolor[7] = usTmp;
VioSetState(pviopal, 0);        /* enables new settings        */
```

See Also

VioCreatePS, VioGetState, VioSetMode

Changes

The **VIOCOLORREG, VIOSETULINELOC,** and **VIOSETTARGET** structures have been added to the list of possible structures for this function.

Corrections

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

■ VioShowBuf — Correction

USHORT VioShowBuf(*offLVB, cbOutput, hvio* **)**

```
USHORT offLVB;     /* offset into logical video buffer */
USHORT cbOutput;   /* length                           */
HVIO hvio;         /* video handle                     */
```

The **VioShowBuf** function updates the physical screen from the logical video buffer (LVB). You may use the logical video buffer to directly manipulate information displayed on the screen.

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

Parameters

offLVB Specifies the offset into the logical video buffer at which the screen update is to start.

cbOutput Specifies the length (in bytes) of the screen area to update.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_INVALID_HANDLE
ERROR_VIO_DETACHED

Comments

If a background process calls **VioShowBuf**, the function will return ERROR_VIO_DETACHED.

Example

This example retrieves the address of the logical video buffer, makes changes to that buffer, and calls **VioShowBuf** to update the physical video buffer from the logical video buffer:

```
PBYTE pbLVB;
USHORT cbOutput;
VioGetBuf((PULONG) &pbLVB, &cbOutput, 0);
    .
    .
    .
VioShowBuf(0,           /* offset into logical video buffer */
    cbOutput,           /* length of screen area            */
    0);                 /* video handle                     */
```

See Also

VioCreatePS, VioGetBuf, VioGetPhysBuf

Corrections

This function is not a family API function.

VioWrtCellStr — Correction

```
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 combinations. A character/attribute combination defines the character to be written and the character attribute by which it is displayed.

If the string is longer than the current line, the function continues writing it at the beginning of the next line, but does not write past the end of the screen.

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

Parameters

pchCellString Points to the cell string to write.

cbCellString Specifies the length (in bytes) of the cell string. The length should be an even number.

usRow Specifies the row at which to start writing the cell string.

usColumn Specifies the column at which to start writing the cell string.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls the **VioWrtCellStr** function to display the string "Hello World!" using 12 different attributes:

```
CHAR achCellString[] = "H\1e\21\31\4o\5 \6W\7o\10r\111\13d\14!";
    .
    .
    .
VioWrtCellStr(achCellString,          /* character/attribute string */
    sizeof(achCellString),            /* length of string           */
    10,                               /* row                        */
    35,                               /* column                     */
    0);                               /* video handle               */
```

See Also

VioCreatePS, VioReadCellStr, VioWrtCharStr, VioWrtTTY

Corrections

The references to cells have been changed to reflect that an attribute can be longer than one byte.

■ VioWrtNCell — Correction

```
USHORT VioWrtNCell(pbCell, cb, usRow, usColumn, hvio)
PBYTE pbCell;         /* pointer to cell to write      */
USHORT cb;            /* number of times to write      */
USHORT usRow;         /* starting position (row)       */
USHORT usColumn;      /* starting position (column)    */
HVIO hvio;            /* video handle                  */
```

The **VioWrtNCell** function writes a cell to the screen a specified number of times. A cell (also called a character/attribute combination) consists of an unsigned byte value that specifies the character and one or more unsigned byte values that specify the attribute to be written.

If the number of times that a cell is repeated is greater than the screen width, the **VioWrtNCell** function continues writing the cell at the beginning of the next line but does not write past the end of the screen.

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

Parameters

pbCell Points to the cell to write.

cb Specifies the number of times to write the cell.

usRow Specifies the row at which to start writing the cell.

usColumn Specifies the column at which to start writing the cell.

hvio Identifies an advanced video-input-and-output (AVIO) presentation space. For AVIO programs, this handle must have been created previously using the **VioCreatePS** function. For other programs, *hvio* must be NULL.

Return Value

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

ERROR_VIO_COL
ERROR_VIO_INVALID_HANDLE
ERROR_VIO_ROW

Example

This example calls the **VioWrtNCell** function to fill the screen with green capital letter A's (on an EGA color monitor):

```
BYTE abCell[2];             /* character/attribute pair */
abCell[0] = 'A';            /* character (letter A)     */
abCell[1] = 0x02;           /* attribute (green)        */
VioWrtNCell(abCell,         /* address of attribute     */
    80 * 25,                /* number of cells to write */
    0,                      /* row                      */
    0,                      /* column                   */
    0);                     /* video handle             */
```

See Also

VioCreatePS, VioWrtNChar

Corrections

The references to cells have been changed to reflect the fact that a attribute can be longer than one byte.

■ WinAddProgram — Change

```
HPROGRAM WinAddProgram(hab, ppib, hGroupHandle)
HAB hab;                    /* handle of anchor block                         */
PPIBSTRUCT ppib;            /* address of structure with program information */
HPROGRAM hGroupHandle;      /* handle of program group                        */
```

The **WinAddProgram** function adds a program to the program list of a group. Program titles need not be unique, although duplicate titles within the same group are not allowed.

Parameters

hab Identifies the anchor block.

ppib Points to a **PIBSTRUCT** structure that contains program information for the program being added to the program list. The **PIBSTRUCT** structure has the following form:

```
typedef struct _PIBSTRUCT {
    PROGTYPE  progt;
    CHAR      szTitle[MAXNAMEL+1];
    CHAR      szIconFileName[MAXPATHL+1];
    CHAR      szExecutable[MAXPATHL+1];
    CHAR      szStartupDir[MAXPATHL+1];
    XYWINSIZE xywinInitial;
    USHORT    res1;
    LHANDLE   res2;
    USHORT    cchEnvironmentVars;
    PCH       pchEnvironmentVars;
    USHORT    cchProgramParameter;
    PCH       pchProgramParameter;
} PIBSTRUCT;
```

hGroupHandle Identifies the program group to which the program is added.

Return Value

The return value is the handle for the program if the function is successful or NULL if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_DUPLICATE_TITLE
PMERR_GROUP_PROTECTED

PMERR_INSUFF_SPACE_TO_ADD
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_PROGRAM_CATEGORY
PMERR_INVALID_TARGET_HANDLE
PMERR_INVALID_TITLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments The **WinAddProgram** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfAddProgram** function.

See Also **PrfAddProgram, WinCreateGroup, WinQueryDefinition, WinQueryProgramTitles**

Changes This function has been replaced by the **PrfAddProgram** function.

■ WinAssociateHelpInstance New

```
BOOL WinAssociateHelpInstance( hwndHelpInstance, hwndApp)
HWND hwndHelpInstance;     /* handle of help instance     */
HWND hwndApp;              /* application-window handle   */
```

The **WinAssociateHelpInstance** function associates a help instance with a specified application window.

Parameters

hwndHelpInstance Identifies the help instance. It must have been previously created using the **WinCreateHelpInstance** function.

hwndApp Identifies the application window with which the help instance is associated, or is NULL. If NULL, the association (if any) between the help instance and a window is removed.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

HMERR_INVALID_ASSOC_HELP_INST
HMERR_INVALID_HELP_INSTANCE_HDL
HMERR_NO_FRAME_WND_IN_CHAIN
HMERR_INVALID_ASSOC_APP_WND

Comments A help instance can be associated with any application window that has a frame, but the help instance should contain help information relating to this application window and the windows in its window chain.

See Also **WinCreateHelpInstance**

WinBroadcastMsg Correction

```
BOOL WinBroadcastMsg( hwnd, msg, mp1, mp2, fs)
HWND hwnd;      /* handle of the parent window  */
USHORT msg;     /* message                      */
MPARAM mp1;     /* message parameter            */
MPARAM mp2;     /* message parameter            */
USHORT fs;      /* windows to send message to   */
```

The **WinBroadcastMsg** function broadcasts a message to multiple windows. This function sends or posts a message to all immediate child windows of the specified window.

Parameters

hwnd Identifies the window whose immediate child windows will receive the message. If this parameter is HWND_DESKTOP, the function sends the message to all main windows on the screen.

msg Specifies the message.

mp1 Specifies the first message parameter.

mp2 Specifies the second message parameter.

fs Specifies which windows to send the message to, and whether the message should be sent or posted. The value consists of a flag from each of the following lists combined using the OR operator.

The following list contains the values specifying which windows to broadcast the message to:

Destination	Meaning
BMSG_DESCENDANTS	Post or send the message to *hwnd* and all of its descendants.
BMSG_FRAMEONLY	Post or send the message to frame windows only.

The following list contains the values specifying how to broadcast the message (send or post):

Value	Meaning
BMSG_POST	Post a message to all child windows of the window specified by the *hwnd* parameter.
BMSG_POSTQUEUE	Post a message to all threads that have a message queue. The message's *hwnd* parameter will be NULL.
BMSG_SEND	Send a message to all children of the window specified by the *hwnd* parameter.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

See Also

WinPostMsg, WinSendMsg

Corrections

To broadcast a message to all windows in the system, the *hwnd* parameter must be set to HWND_DESKTOP, not to NULL.

■ WinCreateFrameControls — Correction

```
BOOL WinCreateFrameControls( hwndFrame, pfcdata, pszTitle, hmod)
HWND hwndFrame;           /* handle of the frame window */
PFRAMECDATA pfcdata;      /* address of structure       */
PSZ pszTitle;             /* address of title-bar string */
```

The **WinCreateFrameControls** function creates standard frame controls for a specified window. This function is used when the standard frame controls are needed for a nonstandard window; for example, with a window with a class other than WC_FRAME.

Parameters

hwndFrame Identifies the frame window that becomes the parent and owner window of all the frame controls created.

pfcdata Points to the **FRAMECDATA** structure that contains information about the frame controls that are to be created. The **FRAMECDATA** structure has the following form:

```
typedef struct _FRAMECDATA {
    USHORT       cb;
    ULONG        flCreateFlags;
    HMODULE      hmodResources;
    USHORT       idResources;
} FRAMECDATA;
```

pszTitle Points to a null-terminated string displayed in a title-bar control.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

See Also

WinCreateWindow

Corrections

The syntax incorrectly listed an *hmod* parameter. This function only has three parameters, not four.

■ WinCreateGroup — Change

```
HPROGRAM WinCreateGroup( hab, pszTitle, fVisible, hprogDest, pszHelp)
HAB hab;                 /* handle of anchor block      */
PSZ pszTitle;            /* address of group title      */
BYTE fVisible;           /* visibility flag             */
HPROGRAM hprogDest;      /* handle of destination group */
PSZ pszHelp;             /* address of help text        */
```

The **WinCreateGroup** function creates a new program-group entry in Desktop Manager. The new group is created empty. The **WinAddProgram** function must be used to add program entries to the group. If the group already exists, the handle of the existing group is returned.

Parameters

hab Identifies the anchor block.

pszTitle Points to the title of the new group. The maximum string size is defined by the MAXNAMEL constant. Strings that exceed this limit are truncated to MAXNAMEL characters. Leading and trailing blanks are removed. The string must contain at least one nonblank character and must not contain a backslash (\).

fVisible Specifies the visibility of the new group. If this parameter is SHE_VISIBLE, the group is visible (it can be viewed by the end-user). If this value is SHE_INVISIBLE, the group is invisible.

hprogDest Identifies the program group into which the new group is placed. If this parameter is NULL, the new group is placed in the root group.

pszHelp Points to a null-terminated string that is used as a short piece of help information relating to the new program group. This parameter is optional and can be NULL. If used, the string must contain at least one nonblank character and be less than 60 characters in length.

Return Value The return value is the group handle for the group if the function is successful. Otherwise, the return value is NULL, indicating that an error occurred.

Errors Use the **WinGetErrorInfo** function to retrieve the error value, which may be one of the following:

PMERR_INSUFF_SPACE_TO_ADD
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_TARGET_HANDLE
PMERR_INVALID_TITLE
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments The **WinCreateGroup** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfCreateGroup** function.

See Also **PrfCreateGroup, WinAddProgram**

Changes This function has been replaced by the **PrfCreateGroup** function.

■ WinCreateHelpInstance New

```
HWND WinCreateHelpInstance( hab, phmInitStructure)
HAB hab;                      /* anchor-block handle    */
PHELPINIT phmInitStructure;   /* pointer to help structure */
```

The **WinCreateHelpInstance** function creates a help instance. A help instance is an "object" window that process help requests from the application and the user.

Parameters *hab* Identifies the application anchor block. It must have been previous creating using the **WinInitialize** function.

phmInitStructure Points to the **HELPINIT** structure. The **HELPINIT** structure has the following form:

```
typedef struct _HELPINIT {
    USHORT     cBytes;
    ULONG      ulReturnCode;
    PSZ        pszTutorialName;
    PHELPTABLE phtHelpTable;
    HMODULE    hmodHelpTableModule;
    HMODULE    hmodAccelActionBarModule;
    USHORT     idAccelTable;
    USHORT     idActionBar;
    PSZ        pszHelpWindowTitle;
    USHORT     usShowPanelId;
    PSZ        pszHelpLibraryName;
} HELPINIT;
```

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

Return Value The return value is the handle of the help instance created if the function is successful or NULL if an error occurs.

See Also **WinCreateHelpTable, WinDestroyHelpInstance, WinInitialize, WinLoadHelpTable**

■ WinCreateHelpTable New

BOOL WinCreateHelpTable (*hwndHelpInstance, phtHelpTable* **)**
HWND *hwndHelpInstance*; /* handle of help instance */
PHELPTABLE *phtHelpTable*; /* pointer to structure with help table */

The **WinCreateHelpTable** function replaces the existing help table (if any) with the help table pointed to by *phtHelpTable*.

Parameters *hwndHelpInstance* Identifies the help instance. It must have been previously created using the **WinCreateHelpInstance** function.

phtHelpTable Points to a **HELPTABLE** structure containing window and corresponding help panel IDs. The **HELPTABLE** structure has the following form:

```
typedef struct _HELPTABLE {
    USHORT        idAppWindow;
    PHELPSUBTABLE phstHelpSubTable;
    USHORT        idExtPanel;
} HELPTABLE;
```

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

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Comments Applications can use this function to replace a help instance's initial help table or to set the table if no initial help table is given. The initial help table is specified in the HELPINIT structure when the help instance is created with the **WinCreateHelpInstance** function. The function replaces the help table without freeing any memory or resources associated with the initial help table.

See Also **WinCreateHelpInstance**, HM_CREATE_HELP_TABLE

■ WinCreatePointerIndirect — New

```
HPOINTER WinCreatePointerIndirect(hwndDesktop, pptri)
HWND hwndDesktop;        /* desktop handle                   */
PPOINTERINFO pptri;      /* pointer to structure with bitmap */
```

The **WinCreatePointerIndirect** function creates a pointer by using the **POINTERINFO** structure. It can create a color pointer.

Parameters

hwndDesktop Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

pptri Points to the **POINTERINFO** structure that contains the bitmap used to create the pointer image. The **POINTERINFO** structure has the following form:

```
typedef struct _POINTERINFO {
    BOOL    fPointer;
    SHORT   xHotspot;
    SHORT   yHotspot;
    HBITMAP hbmPointer;
} POINTERINFO;
```

Return Value

The return value is the handle of the new pointer if successful or NULL if an error occurs.

Comments

The **WinCreatePointerIndirect** and **WinCreatePointer** functions are similar. The difference between them is that the **WinCreatePointerIndirect** function can create a color pointer; the **WinCreatePointer** function can create only a black-and-white pointer.

See Also

WinCreatePointer

■ WinCreateSwitchEntry — New

```
HSWITCH WinCreateSwitchEntry(hab, pswctl)
HAB hab;            /* anchor-block handle                         */
PSWCNTRL pswctl;    /* pointer to structure with new entry information */
```

The **WinCreateSwitchEntry** function creates an entry in the switch list (the list of running programs displayed in the Task List).

Parameters

hab Identifies the anchor block.

pswctl Points to the **SWCNTRL** structure that contains information about the new switch-list entry. If the **szSwtitle** field in the **SWCNTRL** structure is NULL, the system uses the name under which the application was started.

This applies only to the first call to this function for that program (since the program was started). Otherwise, a NULL entry name is invalid. The **SWCNTRL** structure has the following form:

```
typedef struct _SWCNTRL {
    HWND     hwnd;
    HWND     hwndIcon;
    HPROGRAM hprog;
    USHORT   idProcess;
    USHORT   idSession;
    UCHAR    uchVisibility;
    UCHAR    fbJump;
    CHAR     szSwtitle[MAXNAMEL+1];
    BYTE     fReserved;
} SWCNTRL;
```

Return Value

The return value is a handle to the new switch-list entry, or NULL if an error occurs.

Comments

The **WinCreateSwitchEntry** and **WinAddSwitchEntry** functions are similar. The only difference between them is that **WinCreateSwitchEntry** takes an anchor-block handle as the first parameter.

Leading and trailing blanks are removed from the title. The title is truncated to 60 characters.

Example

This example calls **WinQueryWindowProcess** to get the current process identifier (needed for the **SWCNTRL** structure). It then sets up the **SWCTL** structure and calls **WinCreateSwitchEntry** to add the program's name to the Task List.

The returned handle can be used in subsequent calls to **WinChangeSwitchEntry** if the title needs to be changed.

The variables *swctl*, *hswitch*, and *pid* should be global if your application will be calling the **WinChangeSwitchEntry** function to avoid having to set up the structure again.

```
SWCNTRL swctl;
HSWITCH hswitch;
PID pid;
HAB hab;

hab = WinQueryAnchorBlock(hwndFrame);               /* gets anchor block  */
WinQueryWindowProcess(hwndFrame, &pid, NULL);       /* gets process id    */

swctl.hwnd = hwndFrame;                             /* window handle      */
swctl.hwndIcon = NULL;                              /* icon handle        */
swctl.hprog = NULL;                                 /* program handle     */
swctl.idProcess = pid;                              /* process identifier */
swctl.idSession = NULL;                             /* session identifier */
swctl.uchVisibility = SWL_VISIBLE;                  /* visibility         */
swctl.fbJump = SWL_JUMPABLE;                        /* jump indicator     */
swctl.szSwtitle[0] = NULL;                          /* program name       */

hswitch = WinCreateSwitchEntry(hab, &swctl);
```

See Also

WinAddSwitchEntry, WinChangeSwitchEntry, WinRemoveSwitchEntry

WinCreateWindow Change

```
HWND WinCreateWindow( hwndParent, pszClass, pszName, flStyle, x, y, cx, cy, hwndOwner,
                      hwndInsertBehind, id, pCtlData, pPresParams)
HWND hwndParent;          /* parent-window handle                   */
PSZ pszClass;             /* pointer to registered class name       */
PSZ pszName;              /* pointer to window text                 */
ULONG flStyle;            /* window style                           */
SHORT x;                  /* horizontal position of window          */
SHORT y;                  /* vertical position of window            */
SHORT cx;                 /* window width                           */
SHORT cy;                 /* window height                          */
HWND hwndOwner;           /* owner-window handle                    */
HWND hwndInsertBehind;    /* handle to sibling window               */
USHORT id;                /* window identifier                      */
PVOID pCtlData;           /* pointer to buffer                      */
PVOID pPresParams;        /* pointer to structure with pres. params. */
```

The **WinCreateWindow** function creates a new window.

Parameters

hwndParent Specifies the parent window of the new window. Any valid window handle can be used.

pszClass Points to the registered class name. This parameter can be an application-specified name (defined by the **WinRegisterClass** function), the name of a preregistered window class, or a window-class (WC) constant.

pszName Points to window text or other class-specific data. The actual structure of the data is class-specific. This data is usually a null-terminated string and is often displayed in the window.

flStyle Specifies the window style. It can be a combination of one or more of the following values:

Value	Meaning
WS_CLIPCHILDREN	Prevents a window from painting over its child windows.
WS_CLIPSIBLINGS	Prevents a window from painting over its sibling windows.
WS_DISABLED	Disables mouse and keyboard input to the window. It is used to temporarily prevent the user from using the window.
WS_MAXIMIZED	Enlarges the window to the maximum size.
WS_MINIMIZED	Reduces the window to the minimum size.
WS_PARENTCLIP	Prevents a window from painting over its parent window.
WS_SAVEBITS	Saves the image under the window as a bitmap. When the window is moved or hidden, the system restores the image by copying the bits.

Value	Meaning
WS_SYNCPAINT	Causes the window to immediately receive WM_PAINT messages after a part of the window becomes invalid. Unless this style is set, the window receives WM_PAINT messages only when no other message is waiting to be processed.
WS_VISIBLE	Makes the window visible. This window is drawn on the screen unless overlapping windows completely obscure it. Windows without this style are hidden.

x Specifies the horizontal position of the window, relative to the origin of the parent window.

y Specifies the vertical position of the window, relative to the origin of the parent window.

cx Specifies the window width, in pels.

cy Specifies the window height, in pels.

hwndOwner Identifies the owner window.

hwndInsertBehind Identifies the sibling window behind which the specified window is placed. If this parameter is HWND_TOP, the specified window is placed on top of all its sibling windows. If this parameter is HWND_BOTTOM, the specified window is placed behind all its sibling windows. If the *hwndInsert-Behind* parameter is neither HWND_TOP nor HWND_BOTTOM, or it is not a child window of the window identified by *hwndParent*, NULL is returned.

id Specifies the window identifier (a value given by the application that allows a specific child window to be identified). For example, the controls of a dialog box have unique identifiers so that an owner window can distinguish which control has notified it. Window identifiers are also used for frame windows.

pCtlData Points to the buffer that contains class-specific information. This data is passed to the window procedure by the WM_CREATE message.

pPresParams Points to a **PRESPARAMS** structure that contains presentation parameters for the window. This parameter is NULL if there are no presentation parameters. The **PRESPARAMS** structure has the following form:

```
typedef struct _PRESPARAMS {
    ULONG   cb;
    PARAM   aparam[1];
} PRESPARAMS;
```

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

Return Value

The return value is the handle of the window if the function is successful or NULL if an error occurs.

Comments

The **WinCreateWindow** function sends a WM_CREATE message to the window procedure of the window being created, and then sends the WM_ADJUSTWINDOWPOS message before the window is displayed. The values passed are those given to the **WinCreateWindow** function.

The WM_SIZE message is not sent by **WinCreateWindow** while the window is being created. Any required size processing is performed during the processing of the WM_CREATE message.

See Also **WinCreateStdWindow, WinQueryObjectWindow, WinRegisterClass**

Changes The *pPresParams* parameter now points to a **PRESPARAMS** structure.

WinDeleteLibrary New

```
BOOL WinDeleteLibrary( hab, hlib)
HAB  hab;    /* anchor-block handle          */
HLIB hlib;   /* handle of library to be deleted */
```

The **WinDeleteLibrary** function deletes a library previously loaded by the **WinLoadLibrary** function.

Parameters

hab Identifies the anchor block.

hlib Identifies the library to be deleted. This handle must have been created by the **WinLoadLibrary** function.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

See Also **WinLoadLibrary**

WinDeleteProcedure New

```
BOOL WinDeleteProcedure( hab, pfnwpProc)
HAB   hab;         /* anchor-block handle        */
PFNWP pfnwpProc;   /* pointer to window function */
```

The **WinDeleteProcedure** function deletes a procedure that was previously loaded using the **WinLoadProcedure** function.

Parameters

hab Identifies the anchor block.

pfnwpProc Points to the procedure to be deleted. This procedure must have been previously loaded by the **WinLoadProcedure** function.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

See Also **WinDeleteLibrary, WinLoadProcedure**

WinDestroyHelpInstance New

```
BOOL WinDestroyHelpInstance( hwndHelpInstance)
HWND hwndHelpInstance;   /* handle of instance to destroy */
```

The **WinDestroyHelpInstance** function destroys a help instance.

Parameters *hwndHelpInstance* Identifies the help instance to destroy. This handle must have been previously created by using the **WinCreateHelpInstance** function.

Return Value The return value is TRUE if the help instance is successfully destroyed or FALSE if an error occurs.

See Also **WinCreateHelpInstance**

■ WinDrawBitmap — Correction

```
BOOL WinDrawBitmap( hpsDst, hbm, prclSrc, pptlDst, clrFore, clrBack, fs)
HPS hpsDst;          /* handle of the destination presentation space    */
HBITMAP hbm;         /* handle of the bitmap                            */
PRECTL prclSrc;      /* address of structure with rectangle coordinates */
PPOINTL pptlDst;     /* address of structure with bitmap position       */
LONG clrFore;        /* color of the foreground                         */
LONG clrBack;        /* color of the background                         */
USHORT fs;           /* bitmap-drawing flags                            */
```

The **WinDrawBitmap** function draws a bitmap using the current image colors and mixes.

Parameters *hpsDst* Identifies the presentation space in which the bitmap is drawn.

hbm Identifies the bitmap.

prclSrc Points to the **RECTL** data structure that contains the coordinates of the rectangle to be drawn. If this parameter is NULL, the entire bitmap is drawn. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

pptlDst Points to the position of the lower left of the bitmap in the presentation space (in device coordinates).

clrFore Specifies the color of the foreground.

clrBack Specifies the color of the background.

fs Specifies the flags that determine how the bitmap is drawn. It can be one of the following values:

Value	Meaning
DBM_HALFTONE	Use the OR operator to combine the bitmap with an alternating pattern of ones and zeros before drawing it. This flag can be used in conjunction with either DBM_NORMAL or DBM_INVERT.
DBM_IMAGEATTRS	The *clrFore* and *clrBack* parameters are ignored and the image attribute colors already selected in *hpsDst* are used instead.
DBM_INVERT	Draw the bitmap inverted, using ROP_NOTSRCCOPY.

Value	Meaning
DBM_NORMAL	Draw the bitmap normally, using ROP_SRCCOPY.
DBM_STRETCH	The *pptlDst* parameter points to a **RECTL** data structure, representing a rectangle in the destination presentation space to which the bitmap should be stretched.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

See Also **GpiCreateBitmap, GpiLoadBitmap, WinGetSysBitmap**

Corrections The previous documentation incorrectly states that the *pptlDst* parameter was specified in presentation-space coordinates. This parameter is specified in device coordinates.

■ WinEnumDlgItem — Change

```
HWND WinEnumDlgItem( hwndDlg, hwnd, code, fLock)
HWND hwndDlg;    /* handle of the dialog window */
HWND hwnd;       /* handle of the child window  */
USHORT code;     /* dialog item to return       */
BOOL fLock;      /* lock/unlock flag            */
```

The **WinEnumDlgItem** function returns the handle of a dialog item within a dialog window.

Parameters

hwndDlg Identifies the dialog window that contains the dialog item.

hwnd Identifies the child window of the dialog window. This may be an immediate child window or a window lower in the hierarchy, such as a child of a child window.

code Specifies which dialog item to return. This parameter is one of the following values:

Value	Meaning
EDI_FIRSTGROUPITEM	First item in same group.
EDI_FIRSTTABITEM	First item in dialog window with style WS_TABSTOP. The *hwnd* window is ignored.
EDI_LASTGROUPITEM	Last item in same group.
EDI_LASTTABITEM	Last item in dialog box with style WS_TABSTOP. The *hwnd* window is ignored.
EDI_NEXTGROUPITEM	Next item in same group. Wraps to beginning of group when end of group is reached.
EDI_NEXTTABITEM	Next item with style WS_TABSTOP. Wraps around to beginning of dialog-item list when end is reached.
EDI_PREVGROUPITEM	Previous item in same group. Wraps to end of group when start of group is reached.

Value	Meaning
EDI_PREVTABITEM	Previous item with style WS_TABSTOP. Wraps to end of dialog-item list when beginning is reached.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value

The return value is the item handle obtained by this function, specified by the *code* parameter. The window is always an immediate child window of the window identified by the *hwndDlg* parameter.

See Also

WinBeginEnumWindows, WinLockWindow

Changes

The *fLock* parameter is ignored in MS OS/2, version 1.2.

■ WinGetDlgMsg New

```
BOOL WinGetDlgMsg(hwnd, pqmsg)
HWND hwnd;      /* dialog-window handle              */
PQMSG pqmsg;    /* pointer to structure with message */
```

The **WinGetDlgMsg** function retrieves a message intended for a dialog box. This function is used by an application written in a language (for example, COBOL, or FORTRAN) that does not allow the system to call the application's window procedure (this is called a non-reentrant window procedure).

Parameters

hwnd Identifies the dialog window.

pqmsg Points to the **QMSG** structure that contains a message. The **QMSG** structure has the following form:

```
typedef struct _QMSG {
    HWND hwnd;
    USHORT msg;
    MPARAM mp1;
    MPARAM mp2;
    ULONG time;
    POINTL ptl;
} QMSG;
```

Return Value

The return value is TRUE if there is a message for the dialog box, or it is FALSE if the dialog is complete or an error occurs.

Comments

The **WinGetDlgMsg** function allows a language that cannot support window procedures to provide the function of a modal dialog window. The application creates a modeless box dialog by using the **WinCreateDlg** or **WinLoadDlg** functions and then calls **WinGetDlgMsg** to process messages associated with the dialog box. The application should call this function in a loop until it receives a WM_QUIT message. The application should call **WinDefDlgProc** for the messages it does not want rather than dispatching the messages it receives.

To create a window that uses a non-reentrant window procedure, use NULL for the *pfnWndProc* parameter of the **WinRegisterClass** function.

The first time this function is called, the owner of the window specified by *hwnd* is disabled, thereby preventing input into windows other than the dialog box. The owner of the window specified by *hwnd* is enabled when the **WinDismissDlg** function is issued by the application or by the default dialog procedure.

Synchronous messages that would normally go directly to the window procedure will be converted to one of the following messages and retrieved by the **WinGet-DlgMsg** function:

WM_PPAINT
WM_PSETFOCUS
WM_PSYSCOLORCHANGE
WM_PSIZE
WM_PACTIVATE
WM_PCONTROL

See Also **WinCreateDlg, WinDefDlgProc, WinDismissDlg, WinLoadDlg, WinRegister-Class**

■ WinGetNextWindow — Change

```
HWND WinGetNextWindow(henum)
HENUM henum;    /* handle of the enumeration list */
```

The **WinGetNextWindow** function obtains the handle of the next window in a specified enumeration list.

The enumeration list details the window hierarchy at the time **WinBegin-EnumWindows** was called. Enumeration starts with the top-most child window (listed first) and proceeds through the list each time the function is called, until all windows have been enumerated. Once all windows have been enumerated, the function returns NULL. The enumeration then returns to the top of the list and the handle of the top-most child window is returned on the next call.

Parameters *henum* Identifies the enumeration list. This parameter is created by the **Win-BeginEnumWindows** function.

Return Value The return value is the handle of the next window in the enumeration list, or it is NULL if an error occurs.

See Also **WinBeginEnumWindows, WinLockWindow**

Changes This function no longer locks the window.

■ WinGetSysBitmap — Change

```
HBITMAP WinGetSysBitmap(hwndDesktop, ibm)
HWND hwndDesktop;    /* handle of the desktop         */
USHORT ibm;          /* index of the system bitmap */
```

The **WinGetSysBitmap** function returns a handle to one of the standard bitmaps provided by the system. This bitmap can be used for any of the normal bitmap operations. When your application is done with the bitmap, it should free it by calling **GpiDeleteBitmap**.

Parameters

hwndDesktop Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

ibm Specifies the system-bitmap index value. It can be one of the following values:

Value	Meaning
SBMP_BTNCORNERS	Push button corners.
SBMP_CHECKBOXES	Check box/radio button check mark.
SBMP_CHILDSYSMENU	Smaller version of the system menu bitmap to use in child windows.
SBMP_COMBODOWN	Combo-box down arrow.
SBMP_DRIVE	A symbol used by the file system to indicate a disk drive.
SBMP_FILE	A symbol used by the file system to indicate a file.
SBMP_FOLDER	A symbol used by the file system to show subdirectories.
SBMP_MAXBUTTON	Maximize button.
SBMP_MENUATTACHED	A symbol used to indicate that a menu item has an attached hierarchical menu.
SBMP_MENUCHECK	Menu check mark.
SBMP_MINBUTTON	Minimize button.
SBMP_PROGRAM	A symbol used by the file system to indicate that a file is an executable program.
SBMP_RESTOREBUTTON	Restore button.
SBMP_SBDNARROW	Scroll-bar down arrow.
SBMP_SBDNARROWDEP	Scroll-bar down arrow is pressed.
SBMP_SBDNARROWDIS	Scroll-bar down arrow is disabled.
SBMP_SBLFARROW	Scroll-bar left arrow.
SBMP_SBLFARROWDEP	Scroll-bar left arrow is pressed.
SBMP_SBLFARROWDIS	Scroll-bar right arrow is disabled.
SBMP_SBRGARROW	Scroll-bar right arrow.
SBMP_SBRGARROWDEP	Scroll-bar right arrow is pressed.
SBMP_SBRGARROWDIS	Scroll-bar right arrow is disabled.
SBMP_SBUPARROW	Scroll-bar up arrow.
SBMP_SBUPARROWDEP	Scroll-bar up arrow is pressed.
SBMP_SBUPARROWDIS	Scroll-bar up arrow is pressed.
SBMP_SIZEBOX	A symbol used to indicate an area of a window that a user can click to resize the window.
SBMP_SYSMENU	System menu.

Value	Meaning
SBMP_TREEMINUS	A symbol used by the file system to show that an entry in the directory tree contains no more files.
SBMP_TREEPLUS	A symbol used by the file system to show that an entry in the directory tree contains more files.

Return Value The return value is a handle to a bitmap, or it is NULL if an error occurs.

See Also **GpiDeleteBitmap, WinDrawBitmap**

Changes The following system bitmaps have been added:

Value	Meaning
SBMP_SBUPARROWDEP	Scroll-bar up arrow is pressed.
SBMP_SBDNARROWDEP	Scroll-bar down arrow is pressed.
SBMP_SBLFARROWDEP	Scroll-bar left arrow is pressed.
SBMP_SBRGARROWDEP	Scroll-bar right arrow is pressed.
SBMP_SBUPARROWDIS	Scroll-bar up arrow is disabled.
SBMP_SBDNARROWDIS	Scroll-bar down arrow is disabled.
SBMP_SBLFARROWDIS	Scroll-bar right arrow is disabled.
SBMP_SBRGARROWDIS	Scroll-bar right arrow is disabled.
SBMP_COMBODOWN	Combo-box down arrow.

■ WinInstStartApp New

```
HAPP WinInstStartApp (hini, hwndNotifyWindow, cCount, pszApp, pszCmdLine, pData, fsOption)
HINI hini;                  /* initialization-file handle                    */
HWND hwndNotifyWindow;      /* notification-window handle                    */
USHORT cCount;              /* count of elements in the application array   */
PSZ * pszApp;               /* identifier of application                     */
PSZ pszCmdLine;             /* input parameters for application              */
PVOID pData;                /* must be zero                                  */
USHORT fsOptions;           /* option flags                                  */
```

The **WinInstStartApp** function starts an installed application.

Parameters

hini Specifies the handle of the initialization file where the application is found.

hwndNotifyWindow Identifies the window to which a notification message should be sent. If this parameter is NULL, no notification message is sent.

cCount Specifies the number of elements in the array pointed to by the *pszApp* parameter. This value must be 1 or 2.

pszApp Points to an array of pointers which, in turn, point to strings that contain the name of the application and group (if any) where the application is found. The first element of the array points to the application name, the second to the group name.

pszCmdLine Points to the string that contains the command line to be passed to the application.

pData Reserved value; must be zero.

fsOptions Specifies the options to be used to start the application. This parameter can be one of the following values:

Value	Meaning
SAF_INSTALLEDCMDLINE	The command-line parameters in the Task List are used. Any parameters specified by *pszCmdLine* are ignored.
SAF_STARTCHILDAPP	The application is started as a child session of the session from which the **WinInst-StartApp** function is called.

Return Value The return value is a handle to the application started if the function is successful or NULL if an error occurs.

Errors Possible errors may be retrieved with the **WinGetLastError** function, and may be one of the following:

PMERR_INVALID_PARAMETERS
PMERR_INVALID_APPL
PMERR_INVALID_WINDOW
PMERR_CANNOT_START
PMERR_STARTED_IN_BACKGROUND
PMERR_DOS_ERROR
PMERR_NOT_ENOUGH_MEM

See Also **WinTerminateApp**

■ WinIsWindowShowing New

```
BOOL WinIsWindowShowing(hwnd)
HWND hwnd;    /* window handle */
```

The **WinIsWindowShowing** function determines if all or part of a window is currently displayed on the screen. This is in contrast to the **WinIsWindowVisible** function, which returns the actual visibility state of the window rather than its displayed state.

Parameters *hwnd* Identifies the window to be checked.

Return Value The return value is TRUE if any part of the identified window is visible, it is FALSE if no part of the window is visible.

Comments The **WinIsWindowShowing** function also returns FALSE if it is called while the user is moving a window.

See Also **WinIsWindowEnabled, WinIsWindowVisible**

WinLoadHelpTable New

```
BOOL WinLoadHelpTable( hwndHelpInstance, idHelpTable, hmodModule)
HWND    hwndHelpInstance;    /* handle of help instance   */
USHORT  idHelpTable;         /* resource ID for help table */
HMODULE hmodModule;          /* resource-module handle    */
```

The **WinLoadHelpTable** function specifies a help table for the given help instance.

Parameters

hwndHelpInstance Identifies the help instance. The instance must have been previously created using the **WinCreateHelpInstance** function.

idHelpTable Specifies the resource ID of the help table.

hmodModule Identifies the module that contains the help table resource.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Errors

Use the **WinGetLastError** function to retrieve the error value, which may be the following:

HMERR_HELP_INST_CALLED_INVALID

Comments

Applications can use this function to replace a help instance's initial help table or to set the table if no initial help table is given. The initial help table is specified in the HELPINIT structure when the help instance is created with the **WinCreateHelpInstance** function. The function replaces the help table without freeing any memory or resources associated with the initial help table.

See Also

WinCreateHelpInstance, HM_LOAD_HELP_TABLE

WinLoadLibrary New

```
HLIB WinLoadLibrary( hab, pszModName)
HAB hab;              /* anchor-block handle   */
PSZ pszModName;       /* pointer to library name */
```

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

Parameters

hab Identifies the anchor block.

pszModName 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 *.dll* filename extension by default.

Return Value

The return value is the handle of the library module, or it is NULL if an error occurs.

See Also

DosLoadModule, WinDeleteLibrary, WinLoadProcedure

■ WinLoadProcedure — New

```
PFNWP WinLoadProcedure(hab, hlib, pszProcName)
HAB  hab;             /* anchor-block handle       */
HLIB hlib;            /* handle of library         */
PSZ  pszProcName;     /* pointer to procedure name */
```

The **WinLoadProcedure** function loads a window procedure from the specified dynamic-link library.

Parameters

hab Identifies the anchor block.

hlib Specifies the library handle. If this parameter is NULL, the **WinLoadLibrary** function will be called, using the value of the *pszProcName* parameter as the library name.

pszProcName Points to the null-terminated string that specifies the name of the procedure to be loaded.

Return Value

The return value is a pointer to the window procedure, or it is NULL if an error occurs.

See Also

WinDeleteProcedure, WinLoadLibrary

■ WinLockWindow — Change

```
HWND WinLockWindow(hwnd, fLock)
HWND hwnd;     /* window handle    */
BOOL fLock;    /* lock/unlock flag */
```

This function exists for compatibility with MS OS/2, version 1.1. It is not used in MS OS/2 1.2 or later versions.

Changes

This function is not used in MS OS/2 1.2 or later versions.

■ WinQueryActiveWindow — Change

```
HWND WinQueryActiveWindow(hwndDesktop, fLock)
HWND hwndDesktop;     /* desktop handle   */
BOOL fLock;           /* lock/unlock flag */
```

The **WinQueryActiveWindow** function retrieves the active frame window.

Parameters

hwndDesktop Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value

The return value is the handle of the active window if the function is successful; it is NULL if no window was active at the time of the call or the desktop handle is invalid.

Comments

If this function is called while the active window is changing, it may return NULL, indicating that no window was active at the time of the call. Because a

NULL value can also be returned if the *hwndDesktop* handle is invalid, the **WinGetLastError** function must be called to determine if a NULL return value is caused by an invalid *hwndDesktop* handle or because the active window was changing when **WinQueryActiveWindow** was called.

See Also **WinGetLastError, WinLockWindow, WinQueryFocus**

Changes The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

■ WinQueryAnchorBlock New

```
HAB WinQueryAnchorBlock( hwnd)
HWND hwnd;    /* window handle */
```

The **WinQueryAnchorBlock** function retrieves the handle of the anchor block of a window.

Parameters *hwnd* Identifies the window whose anchor-block handle is to be returned.

Return Value The return value is the anchor-block handle of the specified window if the function is successful or NULL if an error occurs.

■ WinQueryCapture Change

```
HWND WinQueryCapture( hwndDesktop, fLock)
HWND hwndDesktop;    /* desktop handle */
BOOL fLock;          /* lock/unlock flag */
```

The **WinQueryCapture** function returns the window handle of the window that has the mouse capture.

Parameters *hwndDesktop* Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop-window handle.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value The return value is the window handle with the mouse capture if the function is successful; it is NULL if no window has the capture or an error occurs.

See Also **WinLockWindow, WinSetCapture**

Changes The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

■ WinQueryClipbrdOwner Change

```
HWND WinQueryClipbrdOwner( hab, fLock)
HAB hab;       /* anchor-block handle   */
BOOL fLock;    /* lock/unlock viewer flag */
```

The **WinQueryClipbrdOwner** function retrieves the handle of the window that currently owns the clipboard (if any).

Parameters

hab Identifies an anchor block.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value

The return value is the window handle of the current clipboard owner if the function is successful; it is NULL if the clipboard is not owned by any window or if an error occurs.

See Also

WinLockWindow, WinQueryClipbrdViewer, WinSetClipbrdOwner

Changes

The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

■ WinQueryClipbrdViewer Change

```
HWND WinQueryClipbrdViewer( hab, fLock)
HAB hab;        /* anchor-block handle    */
BOOL fLock;     /* lock/unlock viewer flag */
```

The **WinQueryClipbrdViewer** function obtains the handle of the current clipboard viewer window (if any).

Parameters

hab Identifies the anchor block.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value

The return value is the handle of the current clipboard viewer window if the function is successful; it is NULL if the clipboard does not have a current viewer window or if an error occurs.

See Also

WinLockWindow, WinQueryClipbrdOwner, WinSetClipbrdViewer

Changes

The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

■ WinQueryDefinition Change

```
USHORT WinQueryDefinition( hab, hProgHandle, ppib, cbMax)
HAB hab;                /* anchor-block handle                        */
HPROGRAM hProgHandle;   /* program handle                             */
PPIBSTRUCT ppib;        /* address of structure for program information */
USHORT cbMax;           /* length of buffer for program information   */
```

The **WinQueryDefinition** function retrieves information about a program or program group.

Parameters

hab Identifies the anchor block.

hProgHandle Identifies the program or group.

ppib Points to a **PIBSTRUCT** structure that receives the program-information data. If the *hProgHandle* parameter is a group handle, only the program-type and program-title fields are significant. The **PIBSTRUCT** structure has the following form:

```
typedef struct _PIBSTRUCT {
    PROGTYPE  progt;
    CHAR      szTitle[MAXNAMEL+1];
    CHAR      szIconFileName[MAXPATHL+1];
    CHAR      szExecutable[MAXPATHL+1];
    CHAR      szStartupDir[MAXPATHL+1];
    XYWINSIZE xywinInitial;
    USHORT    res1;
    LHANDLE   res2;
    USHORT    cchEnvironmentVars;
    PCH       pchEnvironmentVars;
    USHORT    cchProgramParameter;
    PCH       pchProgramParameter;
} PIBSTRUCT;
```

cbMax Specifies the maximum length (in bytes) of data that can be returned in the data structure pointed to by the *ppib* parameter. If this value is zero, the **WinQueryDefinition** function returns the number of bytes in the program-information block.

Return Value The return value is the length of the data actually returned in the data structure, or zero if an error occurs.

If the target is a program rather than a program group, the data returned in the *ppib* parameter is in a format that can be used by the **WinAddProgram** function.

Errors Use the **WinGetErrorInfo** function to retrieve the error value, which may be one of the following:

PMERR_BUFFER_TOO_SMALL
PMERR_INVALID_PROGRAM_HANDLE
PMERR_MEMORY_ALLOCATION_ERR
PMERR_MEMORY_DEALLOCATION_ERR
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments The **WinQueryDefinition** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfQueryDefinition** function.

See Also **PrfQueryDefinition, WinAddProgram**

Changes This function has been replaced by the **PrfQueryDefinition** function.

■ WinQueryFocus — Change

HWND WinQueryFocus(*hwndDesktop, fLock***)**
HWND *hwndDesktop*; /* desktop handle */
BOOL *fLock*; /* lock/unlock flag */

The **WinQueryFocus** function returns the handle of the window that currently has the focus.

Parameters *hwndDesktop* Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value The return value is a handle to the focus window or NULL if there is no focus window or an error occurs.

See Also **WinFocusChange, WinLockWindow, WinQueryActiveWindow, WinSetFocus**

Changes The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

■ WinQueryHelpInstance — New

```
HWND WinQueryHelpInstance( hwndApp)
HWND hwndApp;    /* handle of application window */
```

The **WinQueryHelpInstance** function retrieves the handle of the help instance associated with the given window.

Parameters *hwndApp* Identifies a window for which the associated help instance is queried.

Return Value The return value is the handle of the associated help instance if the function is successful; it is FALSE if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

HMERR_INVALID_QUERY_APP_WND
HMERR_NO_HELP_INST_IN_CHAIN

Comments The function traces the chain of parent windows, starting with the given window, until it finds a frame window with an associated help instance or finds the desktop. If it finds the desktop, it traces the chain of owner windows, starting with the given window, until it finds a frame window with an associated help instance or the desktop.

See Also **WinAssociateHelpInstance**

■ WinQueryPresParam — New

```
ULONG WinQueryPresParam( hwnd, id1, id2, pulId, cbBuf, pbBuf, fs)
HWND hwnd;       /* window handle                              */
ULONG id1;       /* first parameter type to retrieve           */
ULONG id2;       /* second parameter type to retrieve          */
PULONG pulId;    /* pointer to variable for parameter ID       */
ULONG cbBuf;     /* buffer length                              */
PVOID pbBuf;     /* pointer to buffer for presentation parameter */
USHORT fs;       /* flags                                      */
```

The **WinQueryPresParam** function retrieves the presentation parameters for a window.

Parameters *hwnd* Identifies the window that contains the presentation parameters to retrieve.

id1 Identifies the first type of presentation parameter to retrieve. If both the *id1* and *id2* parameters are found, *id1* takes precedence and its presentation parameter is returned. This parameter is ignored if it is zero.

id2 Identifies the second type of presentation parameter to retrieve. If both the *id1* and *id2* parameters are found, *id1* takes precedence and its presentation parameter is returned. This parameter is ignored if it is zero.

pulId Points to the variable that receives the presentation parameter ID.

cbBuf Specifies the length (in bytes) of the buffer pointed to by the *pbBuf* parameter.

pbBuf Points to the buffer that receives the presentation parameter.

fs Specifies one or more flags. These can be any combination of the following values:

Value	Meaning
QPF_NOINHERIT	Specifies that only the window identified by *hwnd* is to be searched for presentation parameters. If this flag is not specified, the entire owner-chain of the window will be searched.
QPF_ID1COLORINDEX	Specifies that the *id1* parameter is a color index. The RGB color equivalent is returned in the *pbBuf* parameter.
QPF_ID2COLORINDEX	Specifies that the *id2* parameter is a color index. The RGB color equivalent is returned in the *pbBuf* parameter.
QPF_PURERGBCOLOR	Specifies that the returned value should be a pure RGB color.

Return Value

The return value is the size (in bytes) of the presentation parameter if the function is successful; it is NULL if no parameter was found or an error occurs.

Comments

The following parameter types are defined for MS OS/2, version 1.2:

Value	Meaning
PP_FOREGROUNDCOLOR	RGB foreground color
PP_FOREGROUNDCOLORINDEX	Color index of foreground color
PP_BACKGROUNDCOLOR	RGB background color
PP_BACKGROUNDCOLORINDEX	Color index of background color
PP_HILITEFOREGROUNDCOLOR	RGB color of foreground highlighted area
PP_HILITEFOREGROUNDCOLORINDEX	Color index of foreground highlighted area
PP_HILITEBACKGROUNDCOLOR	RGB color of background highlighted area
PP_HILITEBACKGROUNDCOLORINDEX	Color index of background highlighted area

Value	Meaning
PP_DISABLEDFOREGROUNDCOLOR	RGB foreground disabled color
PP_DISABLEDFOREGROUNDCOLORINDEX	Color index of foreground disabled color
PP_DISABLEDBACKGROUNDCOLOR	RGB color of background disabled color
PP_DISABLEDBACKGROUNDCOLORINDEX	Color index of background disabled color
PP_BORDERCOLOR	RGB color of window border
PP_BORDERCOLORINDEX	Color index of window border
PP_FONTNAMESIZE	Font size.
PP_FONTHANDLE	Font handle.

See Also **WinSetPresParam**

WinQueryProfileData — Change

```
BOOL WinQueryProfileData( hab, pszAppName, pszKeyName, pvBuf, cbBuf)
HAB hab;              /* anchor-block handle                       */
PSZ pszAppName;       /* address of application name               */
PSZ pszKeyName;       /* address of keyname                        */
PVOID pvBuf;          /* address of buffer                         */
PUSHORT pcbBuf;       /* address of variable with length of buffer */
```

The **WinQueryProfileData** function retrieves binary data from the *os2.ini* file. The location of the data is determined by an application name and a keyname that are passed to the function.

Parameters

hab Identifies an anchor block.

pszAppName Points to a null-terminated string that contains the name of the application. The length of the string must be less than 1024 bytes, including the null terminating character. The application name is case-sensitive. If *pszApp-Name* is NULL, all application names are returned.

pszKeyName Points to a null-terminated string that contains the keyname. The length of the string must be less than 1024 bytes, including the null terminating character. The keyname is case-sensitive. If *pszKeyName* is NULL, all keynames are returned.

pvBuf Points to a buffer that receives the data.

pcbBuf Points to a variable that contains the size of the buffer pointed to by the *pvBuf* parameter. When the function returns, this variable contains the actual number of bytes placed into the buffer.

Return Value The return value is TRUE if the function is successful, or FALSE if an error occurs.

Comments You can find out the size of the data prior to calling this function by calling the **WinQueryProfileSize** function.

The **WinQueryProfileData** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfQueryProfileData** function.

See Also **PrfQueryProfileData, WinQueryProfileSize, WinWriteProfileData**

Changes This function has been replaced by the **PrfQueryProfileData** function.

■ WinQueryProfileInt — Change

```
SHORT WinQueryProfileInt( hab, pszAppName, pszKeyName, sError)
HAB hab;             /* anchor-block handle              */
PSZ pszAppName;      /* address of application name      */
PSZ pszKeyName;      /* address of keyname               */
SHORT sError;        /* value returned if keyname not found */
```

The **WinQueryProfileInt** function retrieves an integer from the *os2.ini* file. The location of the integer is determined by an application name and a keyname which are passed to this function. The **WinWriteProfileString** function must have been used previously to store the integer as a string. For example, a string stored as "123" would be returned as the integer 123. The string may contain a leading minus sign if the number is negative.

Parameters *hab* Identifies the anchor block.

pszAppName Points to a null-terminated string that contains the name of the application. The length of the string must be less than 1024 bytes, including the null terminating character. The application name is case-sensitive.

pszKeyName Points to a null-terminated string that contains the keyname. The length of the string must be less than 1024 bytes, including the null terminating character. The keyname is case-sensitive.

sError Specifies the error value returned if the keyname specified by the *pszKeyName* parameter cannot be found.

Return Value The return value is the integer representation of the text string. If the keyname cannot be found, the error value specified by the *sError* parameter is returned.

Errors The error value may be one of the following:

PMERR_BUFF_TOO_SMALL
PMERR_CAN_NOT_CALL_SPOOLER
PMERR_INVALID_PARM
PMERR_NOT_IN_IDX

Comments The **WinQueryProfileInt** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfQueryProfileInt** function.

See Also **PrfQueryProfileInt, WinQueryProfileData, WinWriteProfileString**

Changes This function has been replaced by the **PrfQueryProfileInt** function.

■ WinQueryProfileSize — Change

```
USHORT WinQueryProfileSize( hab, pszAppName, pszKeyName, pcb)
HAB hab;               /* anchor-block handle                      */
PSZ pszAppName;        /* pointer to application name              */
PSZ pszKeyName;        /* pointer to keyname                       */
PUSHORT pcb;           /* pointer to variable with length of data  */
```

The **WinQueryProfileSize** function retrieves the size of the data stored at a specified location in the *os2.ini* file. The location of the data is determined by an application name and a keyname that are passed to this function. This function is typically called to determine how much memory to allocate prior to calling the **WinQueryProfileData** function.

Parameters

hab Identifies an anchor block.

pszAppName Points to a null-terminated string that contains the name of the application. The length of the string must be less than 1024 bytes, including the null terminating character. The application name is case-sensitive. If *pszAppName* is NULL, the length returned in the variable pointed to by the *pcb* parameter is the length required to contain a list of all application names for the *pszKeyName* parameter.

pszKeyName Points to a null-terminated string that contains the keyname. The length of the string must be less than 1024 bytes, including the null terminating character. The keyname is case-sensitive. If *pszKeyName* is NULL, the length returned in the variable pointed to by the *pcb* parameter is the length required to contain a list of all keynames.

pcb Points to a variable that receives the length of the data. If an error occurs, the length is not returned.

Return Value

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

PMERR_CAN_NOT_CALL_SPOOLER
PMERR_INVALID_PARM
PMERR_NOT_IN_IDX

Comments

The **WinQueryProfileSize** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfQueryProfileSize** function.

See Also

PrfQueryProfileSize, WinQueryProfileData, WinQueryProfileString

Changes

This function has been replaced by the **PrfQueryProfileSize** function.

■ WinQueryProfileString — Change

```
USHORT WinQueryProfileString( hab, pszAppName, pszKeyName, pszError, pszBuf, cchBuf)
HAB hab;              /* anchor-block handle        */
PSZ pszAppName;       /* pointer to application name */
PSZ pszKeyName;       /* pointer to keyname         */
PSZ pszError;         /* pointer to default string  */
PSZ pszBuf;           /* address of buffer for string */
USHORT cchBuf;        /* size of buffer             */
```

The **WinQueryProfileString** function retrieves a string from the *os2.ini* file. The location of the string is determined by an application name and a keyname that are passed to this function.

Parameters

hab Identifies an anchor block.

pszAppName Points to a null-terminated string that contains the name of the application. The length of the string must be less than 1024 bytes, including the null terminating character. The application name is case-sensitive. If the application name is NULL, a list of all applications for the *pszKeyName* parameter is returned.

pszKeyName Points to a null-terminated string that contains the keyname. The length of the string must be less than 1024 bytes, including the null terminating character. If this parameter is NULL, all keynames are enumerated. The keyname is case-sensitive.

pszError Points to a null-terminated string that is placed in the buffer pointed to by the *pszBuf* parameter if the key is not found.

pszBuf Points to a buffer that will receive the null-terminated string.

cchBuf Specifies the length of the buffer pointed to by the *pszBuf* parameter. If the retrieved string is longer than this value, it is truncated.

Return Value The return value is the number of characters in the buffer pointed to by the *pszBuf* parameter.

Comments The **WinQueryProfileString** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfQueryProfileString** function.

See Also **PrfQueryProfileString, WinWriteProfileString**

Changes This function has been replaced by the **PrfQueryProfileString** function.

■ WinQueryProgramTitles — Change

```
USHORT WinQueryProgramTitles( hab, hGroup, paproge, cbBuf, pcTitles)
HAB hab;                     /* handle of anchor block                        */
HPROGRAM hGroup;             /* handle of group                               */
PPROGRAMENTRY paproge;       /* pointer to array of structures for program info. */
USHORT cbBuf;                /* length of buffer for array of structures      */
PUSHORT pcTitles;            /* pointer to variable for number of titles      */
```

The **WinQueryProgramTitles** function obtains information about programs within a specified program group.

You can use the **WinQueryProgramTitles** function to find out the number of entries within a group. If you pass a buffer of zero bytes, the function returns the total number of entries within the group.

The list of returned program entries may contain group handles. Group handles allow the tree structure to be built by the caller; however, this function returns information from only one level of the tree structure.

WinQueryProgramTitles can be used to retrieve the program title, by specifying a program handle in the *hGroup* parameter. In this case, the buffer will contain an entry for only one program.

Parameters

hab Identifies the anchor block.

hGroup Identifies the group for which information is returned. This handle is either the handle of a program group or SGH_ROOT for the root group.

paproge Points to an array of **PROGRAMENTRY** structures where the program information is returned. The **PROGRAMENTRY** structure has the following form:

```
typedef struct _PROGRAMENTRY {
    HPROGRAM hprog;
    PROGTYPE progt;
    CHAR     szTitle[MAXNAMEL+1];
} PROGRAMENTRY;
```

cbBuf Specifies the total length (in bytes) of the area pointed to by the *paproge* parameter. Values of *cbBuf* less than the size of a **PROGRAMENTRY** structure are invalid.

pcTitles Points to a variable that receives the count of the available titles. If the *hGroup* parameter is SGH_ROOT and the buffer length specified in the *cbBuf* parameter is too small to hold all the titles, the return value is zero, none of the titles are copied to the buffer, and *pcTitles* contains the number of available titles. If *hGroup* is a program handle, both the return value and *pcTitles* are the number of available handles.

Return Value

The return value is the number of available titles, or zero if an error occurs.

Errors

Use the **WinGetErrorInfo** function to retrieve the error value, which may be one of the following:

PMERR_BUFFER_TOO_SMALL
PMERR_INVALID_GROUP_HANDLE
PMERR_INVALID_TARGET_HANDLE
PMERR_NOT_CURRENT_PL_VERSION
PMERR_NOT_IN_IDX

Comments

The **WinQueryProgramTitles** function provides compatibility with MS OS/2 1.1 or earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfQueryProgramTitles** function.

See Also

PrfQueryProgramTitles, WinAddProgram

Changes

This function has been replaced by the **PrfQueryProgramTitles** function.

WinQuerySessionTitle New

```
USHORT WinQuerySessionTitle( hab, usSession, pszTitle, cbTitle)
HAB hab;              /* anchor-block handle       */
USHORT usSession;     /* screen session            */
PSZ pszTitle;         /* pointer to buffer for title */
USHORT cbTitle;       /* buffer length             */
```

The **WinQuerySessionTitle** function retrieves the title under which a specified application was started or added to the Task List.

Parameters

hab Identifies the anchor block.

usSession Specifies the screen session. For MS OS/2 version 1.2, this value may be 0 or 1; 0 means the screen session of the caller.

pszTitle Points to the buffer that receives the null-terminated string that specifies the application's title.

cbTitle Specifies the length (in bytes) of the buffer pointed by *pszTitle*. If the title string is longer than this length, the title will be truncated.

Return Value

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

PMERR_INVALID_SESSION_ID

Comments

The length of the title is guaranteed not to exceed MAXNAMEL bytes, plus one for the null-terminating character. (MAXNAMEL is defined in the MS OS/2 include files.)

Example

This example calls **WinQuerySessionTitle** to retrieve the application's title, and then sets the title bar of the frame window to that title:

```
CHAR szTitle[MAXNAMEL + 1];

WinQuerySessionTitle(hab, 0, szTitle, sizeof(szTitle));
WinSetWindowText(hwndFrame, szTitle);
```

See Also

WinSetWindowText

WinQuerySwitchEntry New

```
USHORT WinQuerySwitchEntry( hSwitch, pswctl)
HSWITCH hSwitch;     /* item handle                        */
PSWCNTRL pswctl;     /* point to structure with item data  */
```

The **WinQuerySwitchEntry** function obtains a copy of the Task List data for a specific application.

Parameters

hSwitch Identifies the Task List item.

pswctl Points to the **SWCNTRL** data structure that contains information about the specified Task List item. The **SWCNTRL** structure has the following form:

```
typedef struct _SWCNTRL {
    HWND     hwnd;
    HWND     hwndIcon;
    HPROGRAM hprog;
    USHORT   idProcess;
    USHORT   idSession;
    UCHAR    uchVisibility;
    UCHAR    fbJump;
    CHAR     szSwtitle[MAXNAMEL+1];
    BYTE     fReserved;
} SWCNTRL;
```

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

PMERR_INVALID_SWITCH_HANDLE

See Also **WinQuerySwitchHandle**

■ WinQuerySwitchHandle New

HSWITCH WinQuerySwitchHandle(*hwnd*, *pidProcess* **)**
HWND *hwnd*; /* window handle */
PID *pidProcess*; /* process identifier */

The **WinQuerySwitchHandle** function retrieves the handle of the Task List item of an application.

Parameters *hwnd* Identifies the frame window of the application. This parameter may be zero if the process identifier is specified in the *pidProcess* parameter.

pidProcess Specifies the process identifier. This parameter may be zero if the window handle is specified in the *hwnd* parameter.

Return Value The return value is the Task List handle for the specified application if the function is successful or NULL if an error occurs.

Comments If both a window handle and a process identifier are supplied, they both must apply to the same application.

Example This example calls **WinQuerySwitchHandle** to get the Task List handle of a frame window, and then calls **WinQuerySwitchEntry** to retrieve information about that application:

```
HSWITCH hswitch;
SWCNTRL swctl;

hswitch = WinQuerySwitchHandle(hwndFrame, 0);
WinQuerySwitchEntry(hswitch, &swctl);
```

See Also **WinQuerySwitchEntry**

WinQuerySwitchList — New

```
USHORT WinQuerySwitchList( hab, pswblk, cbswblk)
HAB hab;                /* anchor-block handle          */
PSWBLOCK pswblk;        /* pointer to structure for items */
USHORT cbswblk;         /* structure length             */
```

The **WinQuerySwitchList** function obtains information about the items in the Task List (the list of programs running in the system).

Parameters

hab Identifies the anchor block.

pswblk Points to **SWBLOCK** structure that receives a description of all the items in the Task List. The **SWBLOCK** structure has the following form:

```
typedef struct _SWBLOCK {
    USHORT  cswentry;
    SWENTRY aswentry[1];
} SWBLOCK;
```

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

cbswblk Specifies the size (in bytes) of the **SWBLOCK** structure. This parameter may be zero to retrieve only the number of Task-list items.

Return Value

The return value is the current number of items in the Task List if the function is successful or zero if an error occurs.

Comments

The **SWBLOCK** structure contains an array of **SWENTRY** structures. The first array contains information about the Task List window. The second array contains information about the first program in the Task List.

Example

This example calls **WinQuerySwitchList** to determine the number of items in the Task List, allocates memory for the required buffer, and calls **WinQuerySwitchList** again to fill the buffer with the information about each program in the Task List:

```
USHORT cbItems, cbBuf;
PSWBLOCK pswblk;
SEL sel;

cbItems = WinQuerySwitchList(hab, NULL, 0);  /* gets num. of items */
cbBuf = (cbItems * sizeof(SWENTRY)) + sizeof(HSWITCH);
DosAllocSeg(cbBuf, &sel, SEG_NONSHARED);     /* allocates buffer   */
pswblk = MAKEP(sel, 0);
WinQuerySwitchList(hab, pswblk, cbBuf);      /* gets struct. array */
```

See Also

WinQuerySwitchEntry

WinQuerySysModalWindow — Change

```
HWND WinQuerySysModalWindow( hwndDesktop, fLock)
HWND hwndDesktop;   /* handle of the desktop */
BOOL fLock;         /* lock/unlock flag      */
```

The **WinQuerySysModalWindow** function returns the current system modal window.

Parameters

hwndDesktop Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value The return value is the handle of the current system modal window. If there is none, the return value is NULL.

See Also **WinLockWindow, WinSetSysModalWindow**

Changes The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

WinQuerySysValue — Change

```
LONG WinQuerySysValue( hwndDesktop, iSysValue)
HWND hwndDesktop;    /* handle of desktop       */
SHORT iSysValue;     /* system value to retrieve */
```

The **WinQuerySysValue** function retrieves a specified system value.

Parameters *hwndDesktop* Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

iSysValue Specifies the system value.

Return Value The return value is the system value if the function is successful, or zero if an error occurs.

Comments The system values can be any of the following values:

Value	Meaning
SV_CMOUSEBUTTONS	Specifies the number of mouse buttons: 1, 2, or 3.
SV_MOUSEPRESENT	Specifies whether the mouse is present. A value of TRUE means the mouse is present.
SV_SWAPBUTTON	Specifies whether the mouse buttons are swapped. A value of TRUE means the mouse buttons are swapped.
SV_CXDBLCLK	Specifies the horizontal spacing for a mouse double-click. When the horizontal distance between two mouse clicks is less than this value, the horizontal spacing requirement for considering two mouse clicks a double-click is met.
SV_CYDBLCLK	Specifies the vertical spacing for a mouse double-click. When the vertical distance between two mouse clicks is less than this value, the vertical spacing requirement for considering two mouse clicks a double-click is met.
SV_DBLCLKTIME	Specifies the mouse double-click time, in milliseconds. When the time between two mouse clicks is less than this value, the temporal requirement for considering two mouse clicks a double-click is met.

Value	Meaning
SV_CXSIZEBORDER	Specifies the number of pels along the *x*-axis in a window-sizing border.
SV_CYSIZEBORDER	Specifies the number of pels along the *y*-axis in a window-sizing border.
SV_ALARM	Specifies whether a call to the **WinAlarm** function generates a sound. A value of TRUE means sound is generated.
SV_CURSORRATE	Specifies the rate at which the cursor blinks, in milliseconds. The blink rate is the time that the cursor remains visible or invisible. Twice this value is the time the cursor takes to cycle from visibility to invisibility and back.
SV_FIRSTSCROLLRATE	Specifies the delay (in milliseconds) between clicking and holding down the mouse button (when the mouse pointer is on a scroll arrow or scroll bar) and the beginning of scroll-bar autorepeat activity.
SV_SCROLLRATE	Specifies the delay (in milliseconds) between scroll-bar autorepeat events.
SV_NUMBEREDLISTS	Reserved.
SV_ERRORFREQ	Specifies the frequency (in hertz) of a **WinAlarm** function WA_ERROR sound.
SV_NOTEFREQ	Specifies the frequency (in hertz) of a **WinAlarm** function WA_NOTE sound.
SV_WARNINGFREQ	Specifies the frequency (in hertz) of a **WinAlarm** function WA_WARNING sound.
SV_ERRORDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** function WA_ERROR sound.
SV_NOTEDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** function WA_NOTE sound.
SV_WARNINGDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** function WA_WARNING sound.
SV_CXSCREEN	Specifies the number of pels along the screen's *x*-axis.
SV_CYSCREEN	Specifies the number of pels along the screen's *y*-axis.
SV_CXVSCROLL	Specifies the number of pels along the *x*-axis of a vertical scroll bar.
SV_CYHSCROLL	Specifies the number of pels along the *y*-axis of a horizontal scroll bar.
SV_CXHSCROLLARROW	Specifies the number of pels along the *x*-axis of a horizontal scroll arrow.

Value	Meaning
SV_CYVSCROLLARROW	Specifies the number of pels along the *y*-axis of a vertical scroll arrow.
SV_CXBORDER	Specifies the number of pels along the *x*-axis of a window border.
SV_CYBORDER	Specifies the number of pels along the *y*-axis of a window border.
SV_CXDLGFRAME	Specifies the number of pels along the *x*-axis of a dialog-box frame.
SV_CYDLGFRAME	Specifies the number of pels along the *y*-axis of a dialog-box frame.
SV_CYTITLEBAR	Specifies the number of pels along the *y*-axis of a title-bar window.
SV_CXHSLIDER	Specifies the number of pels along the *x*-axis of a horizontal scroll-bar slider.
SV_CYVSLIDER	Specifies the number of pels along the *y*-axis of a vertical scroll-bar slider.
SV_CXMINMAXBUTTON	Specifies the width (in pels) of a minimize or maximize button.
SV_CYMINMAXBUTTON	Specifies the height (in pels) of a minimize or maximize button.
SV_CYMENU	Specifies the height (in pels) of a menu.
SV_CXFULLSCREEN	Specifies the number of pels along the *x*-axis of the client window of a maximized frame window.
SV_CYFULLSCREEN	Specifies the number of pels along the *y*-axis of the client window of a maximized frame window.
SV_CXICON	Specifies the number of pels along an icon's *x*-axis.
SV_CYICON	Specifies the number of pels along an icon's *y*-axis.
SV_CXPOINTER	Specifies the number of pels along the mouse pointer's *x*-axis.
SV_CYPOINTER	Specifies the number of pels along the mouse pointer's *y*-axis.
SV_DEBUG	Specifies whether a debugging version of OS/2 is being run. This value is TRUE if a debugging version is being run.
SV_CURSORLEVEL	Specifies the cursor display count. The cursor is visible only when the display count is zero.
SV_POINTERLEVEL	Specifies the mouse-pointer display count. The mouse is visible only when the display count is zero.

Value	Meaning
SV_TRACKRECTLEVEL	Specifies the tracking-rectangle display count. The tracking rectangle is visible only when the display count is zero.
SV_CTIMERS	Specifies the number of available timers.
SV_CXBYTEALIGN	Specifies a horizontal alignment that is more efficient for the device driver.
SV_CYBYTEALIGN	Specifies a vertical alignment that is more efficient for the device driver.
SV_EXTRAKEYBEEP	Specifies whether beep is turned on for extended keys (keys not on an IBM PS/2 or compatible keyboard).
SV_SETLIGHTS	Specifies if the system controls the keyboard indicator lights.
SV_INSERTMODE	Specifies if insert mode is on or off for entry-field controls.
SV_MENUROLLDOWNDELAY	Specifies the delay for menu roll down.
SV_MENUROLLUPDELAY	Specifies the delay for menu roll up.
SV_ALTMNEMONIC	Specifies if the Alt key is allowed as a mnemonic.
SV_TASKLISTMOUSEACCESS	Specifies if the task list can be accessed by the right mouse button.
SV_CSYSVALUES	Specifies the number of system values.

See Also **WinSetSysValue**

Changes The following system values have been added:

SV_EXTRAKEYBEEP
SV_SETLIGHTS
SV_INSERTMODE
SV_MENUROLLDOWNDELAY
SV_MENUROLLUPDELAY
SV_ALTMNEMONIC
SV_TASKLISTMOUSEACCESS

WinQueryTaskSizePos New

```
USHORT WinQueryTaskSizePos( hab, usSession, pswp)
HAB hab;              /* anchor-block handle          */
USHORT usSession;     /* screen session               */
PSWP pswp;            /* pointer to structure for defaults */
```

The **WinQueryTaskSizePos** function retrieves the default size, position, and status for the first frame window of a newly started application.

Parameters *hab* Identifies the anchor block.

usSession Specifies the screen session. For MS OS/2 version 1.2, this value can be 0 or 1; 0 specifies the screen session of the caller.

pswp Points to the **SWP** structure that receives the default size, position, and status for the first frame window of the application. The **SWP** structure has the following form:

```
typedef struct _SWP {
    USHORT fs;
    SHORT  cy;
    SHORT  cx;
    SHORT  y;
    SHORT  x;
    HWND   hwndInsertBehind;
    HWND   hwnd;
} SWP;
```

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

PMERR_INVALID_SESSION_ID

See Also **WinQueryWindowPos**

WinQueryWindow Change

```
HWND WinQueryWindow(hwnd, cmd, fLock)
HWND hwnd;      /* handle of the window       */
SHORT cmd;      /* which window to retrieve   */
BOOL fLock;     /* lock/unlock flag           */
```

The **WinQueryWindow** function retrieves the handle of a window that has a specified relationship to a specified window.

If **WinQueryWindow** is used to enumerate windows of other threads, it is not guaranteed that all the windows are enumerated, because the Z ordering of the windows may change during the enumeration. The **WinGetNextWindow** function must be used for this purpose.

Parameters *hwnd* Identifies a window. The window handle retrieved is relative to this window, based on the value in the *cmd* parameter.

cmd Specifies which window to retrieve. The following are the possible values:

Value	Meaning
QW_NEXT	Next window in Z order (window below).
QW_PREV	Previous window in Z order (window above).
QW_TOP	Topmost child window.
QW_BOTTOM	Bottommost child window.
QW_OWNER	Owner of the window.
QW_PARENT	Parent of window; HWND_OBJECT if object window.

Value	Meaning
QW_NEXTTOP	Next main window in the enumeration order defined for the ALT+ESCAPE function of the user interface.
QW_PREVTOP	Previous main window, in the enumeration order defined by QW_NEXTTOP.
QW_FRAMEOWNER	Returns the owner of *hwnd*, normalized so that it shares the same parent as *hwnd*.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value The return value is the handle of the window related to the window identified by the *hwnd* parameter.

See Also **WinGetNextWindow, WinLockWindow**

Changes The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

WinQueryWindowLockCount — Change

```
SHORT WinQueryWindowLockCount(hwnd)
HWND hwnd;    /* window handle */
```

This function exists for compatibility with MS OS/2 version 1.1. It is not used in MS OS/2 1.2 or later versions.

Changes This function is not used in MS OS/2 1.2 or later versions.

WinRegisterClass — Change

```
BOOL WinRegisterClass(hab, pszClassName, pfnWndProc, flStyle, cbWindowData)
HAB hab;                  /* handle of anchor block        */
PSZ pszClassName;         /* points to class name          */
PFNWP pfnWndProc;         /* address of window procedure   */
ULONG flStyle;            /* window-style flags            */
USHORT cbWindowData;      /* amount of reserved data       */
```

The **WinRegisterClass** function registers a window class.

When an application registers a private class with the window procedure in a dynamic-link library, the application must resolve the window-procedure address before calling **WinRegisterClass**.

Private classes are deleted when the process that registers them terminates.

Parameters *hab* Identifies the anchor block.

pszClassName Points to a null-terminated string that specifies the name of the window class. The string can be either a name specified by an application or the name of one of the following preregistered classes:

Class	Description
WC_BUTTON	A button control, including push buttons, radio buttons, check boxes, and user buttons.
WC_COMBOBOX	A combination entry-field and list-box control.
WC_ENTRYFIELD	An entry-field control that allows single-line text editing.
WC_FRAME	A standard frame window.
WC_LISTBOX	A list box that displays items in a list that can be scrolled.
WC_MLE	A multiple-line entry field.
WC_MENU	A menu, including the menu bar and the menus that can selected from it.
WC_SCROLLBAR	A scroll bar that allows a user to scroll the contents of a window.
WC_STATIC	A static control that displays text, icon, or bitmap data.
WC_TITLEBAR	A title-bar control that displays the title of a window across the top of the frame and also allows the user to drag the frame window to a new location.

pfnWndProc Points to the window procedure. This value can be NULL if the application does not provide a window procedure. An application written in a language that does not allow the system to call the application's window procedure (for example, COBOL or FORTRAN) should also use NULL for this parameter. For more information, see **WinGetDlgMsg**.

flStyle Specifies the default window style, which can be any of the standard CS class styles, and any class-specific window styles that may be defined. These styles can be augmented when a window of this class is created. A public window class is created if the CS_PUBLIC style is specified; otherwise, a private class is created. Public classes are available from any process for creating a window. Private classes are available only to the registering process.

The following list describes the standard classes:

Style	Meaning
CS_CLIPCHILDREN	Sets the WS_CLIPCHILDREN style for windows created using this class.
CS_CLIPSIBLINGS	Sets the WS_CLIPSIBLINGS style for windows created using this class.
CS_FRAME	Identifies windows created using this class as frame windows.
CS_HITTEST	Directs the system to send a WM_HITTEST message to a window of this class whenever the mouse moves in the window.
CS_MOVENOTIFY	Directs the system to send a WM_MOVE message to the window whenever the window moves.

Style	Meaning
CS_PARENTCLIP	Sets the WS_PARENTCLIP style for windows created using this class.
CS_PUBLIC	Creates a public window class.
CS_SAVEBITS	Sets the WS_SAVEBITS style for windows created using this class.
CS_SIZEREDRAW	Directs the system to invalidate the entire window whenever the size of the window changes.
CS_SYNCPAINT	Sets the WS_SYNCPAINT style for windows created using this class.

cbWindowData Specifies the number of bytes of storage reserved for use by applications for each window created of this class.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Example

This example calls **WinRegisterClass** to register a class or returns FALSE if an error occurs.

```
HAB hab;
CHAR szClassName[] = "Generic"; /* window class name      */

if (!WinRegisterClass(hab,    /* anchor-block handle     */
        szClassName,          /* class name              */
        GenericWndProc,       /* window procedure        */
        0L,                   /* window style            */
        0))                   /* amount of reserved memory */
    return (FALSE);
```

See Also

WinGetDlgMsg, WinQueryClassInfo, WinQueryClassName, WinQueryWindowPtr, WinQueryWindowULong, WinQueryWindowUShort

Changes

The constants WC_COMBOBOX and WC_MLE have been added to the list of preregistered classes.

I WinReleasePS — Correction

BOOL WinReleasePS(*hps* **)**
HPS *hps*; /* presentation-space handle */

The **WinReleasePS** function releases a cached presentation space obtained by using the **WinGetClipPS, WinGetPS,** or **WinGetScreenPS** function.

Only a cached presentation space can be released using this function. The presentation space is returned to the cache for reuse. The presentation-space handle should not be used following this function.

Parameters

hps Identifies the cached presentation space to release.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Comments

Before an application terminates, it must call **WinReleasePS** to release any cached presentation spaces obtained.

Example

This example processes an application-defined message (IDM_FILL). It calls **WinGetPS** to get a presentation space to the entire window. It gets the dimensions of the current window, fills the window, and calls **WinReleasePS** to release the presentation space.

```
case IDM_FILL:
    hps = WinGetPS(hwnd);                  /* gets ps for entire window  */
    WinQueryWindowRect(hwnd, &rcl);        /* gets window dimensions     */
    WinFillRect(hps, &rcl, CLR_WHITE);     /* clears entire window       */
    WinReleasePS(hps);                     /* releases ps                */
```

See Also

WinGetClipPS, WinGetPS, WinGetScreenPS

Corrections

The **WinReleasePS** function is used to release any cached presentation space, including those created with the **WinGetClipPS** and **WinGetScreenPS** functions.

■ WinRemovePresParam — New

```
BOOL WinRemovePresParam(hwnd, id)
HWND hwnd;     /* window handle                      */
ULONG id;      /* presentation parameter to remove   */
```

The **WinRemovePresParam** function removes a presentation parameter.

Parameters

hwnd Identifies the window that contains the presentation parameters to remove.

id Identifies the presentation parameter to remove. It may be one of the following values:

Value	Meaning
PP_FOREGROUNDCOLOR	RGB foreground color
PP_FOREGROUNDCOLORINDEX	Color index of foreground color
PP_BACKGROUNDCOLOR	RGB background color
PP_BACKGROUNDCOLORINDEX	Color index of background color
PP_HILITEFOREGROUNDCOLOR	RGB color of foreground highlighted area
PP_HILITEFOREGROUNDCOLORINDEX	Color index of foreground highlighted area
PP_HILITEBACKGROUNDCOLOR	RGB color of background highlighted area
PP_HILITEBACKGROUNDCOLORINDEX	Color index of background highlighted area
PP_DISABLEDFOREGROUNDCOLOR	RGB foreground disabled color
PP_DISABLEDFOREGROUNDCOLORINDEX	Color index of foreground disabled color

Value	Meaning
PP_DISABLEDBACKGROUNDCOLOR	RGB color of background disabled color
PP_DISABLEDBACKGROUNDCOLORINDEX	Color index of background disabled color
PP_BORDERCOLOR	RGB color of window border
PP_BORDERCOLORINDEX	Color index of window border
PP_FONTNAMESIZE	Font size
PP_FONTHANDLE	Font handle

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Comments When a presentation parameter is removed, a WM_PRESPARAMCHANGED message is sent to all windows owned by the window calling the **WinSetPresParam** function.

See Also **WinQueryPresParam, WinSetPresParam**

■ WinSetPresParam New

```
BOOL WinSetPresParam( hwnd, id, cbParam, pbParam)
HWND hwnd;          /* window handle                    */
ULONG id;           /* presentation parameter           */
ULONG cbParam;      /* presentation-parameter size      */
PVOID pbParam;      /* pointer to presentation parameter */
```

The **WinSetPresParam** function sets a presentation parameter.

Parameters *hwnd* Identifies the window that contains the presentation parameters to set.

id Identifies the presentation parameter to set. It may be one of the following values:

Value	Meaning
PP_FOREGROUNDCOLOR	RGB foreground color
PP_FOREGROUNDCOLORINDEX	Color index of foreground color
PP_BACKGROUNDCOLOR	RGB background color
PP_BACKGROUNDCOLORINDEX	Color index of background color
PP_HILITEFOREGROUNDCOLOR	RGB color of foreground highlighted area
PP_HILITEFOREGROUNDCOLORINDEX	Color index of foreground highlighted area
PP_HILITEBACKGROUNDCOLOR	RGB color of background highlighted area

Value	Meaning
PP_HILITEBACKGROUNDCOLORINDEX	Color index of background highlighted area
PP_DISABLEDFOREGROUNDCOLOR	RGB foreground disabled color
PP_DISABLEDFOREGROUNDCOLORINDEX	Color index of foreground disabled color
PP_DISABLEDBACKGROUNDCOLOR	RGB color of background disabled color
PP_DISABLEDBACKGROUNDCOLORINDEX	Color index of background disabled color
PP_BORDERCOLOR	RGB color of window border
PP_BORDERCOLORINDEX	Color index of window border
PP_FONTNAMESIZE	Font size
PP_FONTHANDLE	Font handle

cbParam Specifies the length (in bytes) of the buffer pointed to by the *pbParam* parameter.

pbParam Points to the buffer that contains the presentation parameter.

Return Value The return value is TRUE if the function is successful or FALSE if an error occurs.

Comments When a presentation parameter is set, a WM_PRESPARAMCHANGED message is sent to all windows owned by the window calling the **WinSetPresParam** function.

See Also **WinQueryPresParam, WinRemovePresParam**

■ WinSetSysColors — Correction

```
BOOL WinSetSysColors( hwndDesktop, flOptions, flFormat, clrFirst, cclr, pclr)
HWND hwndDesktop;       /* handle of the desktop         */
ULONG flOptions;        /* color options                 */
ULONG flFormat;         /* format options                */
COLOR clrFirst;         /* first color to set            */
ULONG cclr;             /* number of colors to set       */
PCOLOR pclr;            /* address of color definitions  */
```

The **WinSetSysColors** function sets system color values. This function sends a WM_SYSCOLORCHANGE message to all main windows in the system to indicate that the colors have changed. When this message is received, applications that depend on the system colors can query the new color values by using the **WinQuerySysColor** function.

After the WM_SYSCOLORCHANGE messages are sent, all windows in the system are invalidated so that they will be redrawn with the new system colors.

WinSetSysColors does not write any system color changes to the *os2.ini* file.

Parameters

hwndDesktop Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

flOptions Specifies the following options:

Value	Meaning
LCOL_PURECOLOR	Indicates that color dithering should not be used to create colors not available in the physical palette. If this option is set, only pure colors will be used and no dithering will be done.
LCOL_RESET	Indicates that the system colors are all to be reset to default before processing the remainder of the data in this function.

flFormat Specifies the format of entries in the table, as follows:

Value	Meaning
LCOLF_CONSECRGB	Array of RGB values that correspond to color indexes. Each entry is 4 bytes.
LCOLF_INDRGB	Array of (index, RGB) values. Each pair of entries is 8 bytes (4 bytes index and 4 bytes color value).

clrFirst Specifies the starting system color index (this parameter is only relevant for the LCOLF_CONSECRGB format). The following system color indexes are defined (each successive index is one larger than its predecessor):

Value	Meaning
SYSCLR_BUTTONLIGHT	Light button
SYSCLR_BUTTONMIDDLE	Middle button
SYSCLR_BUTTONDARK	Dark button
SYSCLR_BUTTONDEFAULT	Default button
SYSCLR_TITLEBOTTOM	Bottom title
SYSCLR_SHADOW	Shadow
SYSCLR_ICONTEXT	Icon text
SYSCLR_DIALOGBACKGROUND	Dialog-box background
SYSCLR_HILITEFOREGROUND	Foreground hilight
SYSCLR_HILITEBACKGROUND	Background hilight
SYSCLR_INACTIVETITLETEXTBGND	Inactive title-text background
SYSCLR_ACTIVETITLETEXTBGND	Active title-text background
SYSCLR_INACTIVETITLETEXT	Inactive title-text
SYSCLR_ACTIVETITLETEXT	Active title-text
SYSCLR_OUTPUTTEXT	Output text
SYSCLR_WINDOWSTATICTEXT	Static text
SYSCLR_SCROLLBAR	Scroll bar
SYSCLR_BACKGROUND	Screen background
SYSCLR_ACTIVETITLE	Title bar of active window
SYSCLR_INACTIVETITLE	Title bar of inactive window

Value	Meaning
SYSCLR_MENU	Menu background
SYSCLR_WINDOW	Window background
SYSCLR_WINDOWFRAME	Window border line
SYSCLR_MENUTEXT	Menu text
SYSCLR_WINDOWTEXT	Window text
SYSCLR_TITLETEXT	Title text
SYSCLR_ACTIVEBORDER	Border fill of active window
SYSCLR_INACTIVEBORDER	Border fill of inactive window
SYSCLR_APPWORKSPACE	Background of certain main windows
SYSCLR_HELPBACKGROUND	Background of help panels
SYSCLR_HELPTEXT	Help text
SYSCLR_HELPHILITE	Highlight of help text

cclr Specifies the number of elements supplied in *pclr*. This parameter may be zero if, for example, the color table is merely to be reset to the default. For LCOLF_INDRGB, this parameter must be an even number. The constant SYSCLR_CSYSCOLORS is set to the total number of system colors.

pclr Specifies the start address of the application data area containing the color-table definition data. The format depends on the value of the *flFormat* parameter. Each color value is a 4-byte integer. The low byte is the blue intensity value (0x000000FF), the second byte is the green intensity value (0x0000FF00), and the third byte is the red intensity value (0x00FF0000). The intensity for each color may range between 0 and 255.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

See Also

WinQuerySysColor

Corrections

The following system colors have been added:

SYSCLR_BUTTONLIGHT
SYSCLR_BUTTONMIDDLE
SYSCLR_BUTTONDARK
SYSCLR_BUTTONDEFAULT
SYSCLR_TITLEBOTTOM
SYSCLR_SHADOW
SYSCLR_ICONTEXT
SYSCLR_DIALOGBACKGROUND
SYSCLR_HILITEFOREGROUND
SYSCLR_HILITEBACKGROUND
SYSCLR_INACTIVETITLETEXTBGND
SYSCLR_ACTIVETITLETEXTBGND
SYSCLR_INACTIVETITLETEXT
SYSCLR_ACTIVETITLETEXT
SYSCLR_OUTPUTTEXT

The system colors were listed alphabetically instead of by numerical order. The numerical order is important because it is used to determine the starting color to change when LCOLF_CONSECRGB is specified for *flFormat*.

■ WinSetSysValue — Change

```
BOOL WinSetSysValue( hwndDesktop, iSysValue, lValue)
HWND hwndDesktop;     /* handle of desktop window */
SHORT iSysValue;      /* system value to change   */
LONG lValue;          /* new system value         */
```

The **WinSetSysValue** function sets the system value.

Parameters

hwndDesktop Identifies the desktop window. This parameter can be HWND_DESKTOP or the desktop window handle.

iSysValue Specifies the system value. For a complete list of possible system values, see the following "Comments" section.

lValue Specifies the system value. Durations are in milliseconds. Frequencies are in hertz; valid values are 0x0025 through 0x7FFF.

Return Value

The return value is TRUE if the system value is successfully set. Otherwise, it is FALSE, indicating that an error occurred.

Comments

The system values can be any of the following values:

Value	Meaning
SV_CMOUSEBUTTONS	Specifies the number of mouse buttons: 1, 2, or 3.
SV_MOUSEPRESENT	Specifies whether the mouse is present. A value of TRUE means the mouse is present.
SV_SWAPBUTTON	Specifies whether the mouse buttons are swapped. A value of TRUE means the mouse buttons are swapped.
SV_CXDBLCLK	Specifies the horizontal spacing for a mouse double-click. When the horizontal distance between two mouse clicks is less than this value, the horizontal spacing requirement for considering two mouse clicks a double-click is met.
SV_CYDBLCLK	Specifies the vertical spacing for a mouse double-click. When the vertical distance between two mouse clicks is less than this value, the vertical spacing requirement for considering two mouse clicks a double-click is met.
SV_DBLCLKTIME	Specifies the mouse double-click time, in milliseconds. When the time between two mouse clicks is less than this value, the temporal requirement for considering two mouse clicks a double-click is met.
SV_CXSIZEBORDER	Specifies the number of pels along the *x*-axis in a window-sizing border.
SV_CYSIZEBORDER	Specifies the number of pels along the *y*-axis in a window-sizing border.

Value	Meaning
SV_ALARM	Specifies whether a call to the **WinAlarm** function generates a sound. A value of TRUE means sound is generated.
SV_CURSORRATE	Specifies the rate at which the cursor blinks, in milliseconds. The blink rate is the time that the cursor remains visible or invisible. Twice this value is the time the cursor takes to cycle from visibility to invisibility and back.
SV_FIRSTSCROLLRATE	Specifies the delay (in milliseconds) between clicking and holding down the mouse button (when the mouse pointer is on a scroll arrow or scroll bar) and the beginning of scroll-bar autorepeat activity.
SV_SCROLLRATE	Specifies the delay (in milliseconds) between scroll-bar autorepeat events.
SV_NUMBEREDLISTS	Reserved.
SV_ERRORFREQ	Specifies the frequency (in hertz) of a **WinAlarm** function WA_ERROR sound.
SV_NOTEFREQ	Specifies the frequency (in hertz) of a **WinAlarm** function WA_NOTE sound.
SV_WARNINGFREQ	Specifies the frequency (in hertz) of a **WinAlarm** function WA_WARNING sound.
SV_ERRORDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** function WA_ERROR sound.
SV_NOTEDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** function WA_NOTE sound.
SV_WARNINGDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** function WA_WARNING sound.
SV_CXSCREEN	Specifies the number of pels along the screen's *x*-axis.
SV_CYSCREEN	Specifies the number of pels along the screen's *y*-axis.
SV_CXVSCROLL	Specifies the number of pels along the *x*-axis of a vertical scroll bar.
SV_CYHSCROLL	Specifies the number of pels along the *y*-axis of a horizontal scroll bar.
SV_CXHSCROLLARROW	Specifies the number of pels along the *x*-axis of a horizontal scroll arrow.
SV_CYVSCROLLARROW	Specifies the number of pels along the *y*-axis of a vertical scroll arrow.
SV_CXBORDER	Specifies the number of pels along the *x*-axis of a window border.

Value	Meaning
SV_CYBORDER	Specifies the number of pels along the *y*-axis of a window border.
SV_CXDLGFRAME	Specifies the number of pels along the *x*-axis of a dialog-box frame.
SV_CYDLGFRAME	Specifies the number of pels along the *y*-axis of a dialog-box frame.
SV_CYTITLEBAR	Specifies the number of pels along the *y*-axis of a title-bar window.
SV_CXHSLIDER	Specifies the number of pels along the *x*-axis of a horizontal scroll-bar slider.
SV_CYVSLIDER	Specifies the number of pels along the *y*-axis of a vertical scroll-bar slider.
SV_CXMINMAXBUTTON	Specifies the width (in pels) of a minimize or maximize button.
SV_CYMINMAXBUTTON	Specifies the height (in pels) of a minimize or maximize button.
SV_CYMENU	Specifies the height (in pels) of a menu.
SV_CXFULLSCREEN	Specifies the number of pels along the *x*-axis of the client window of a maximized frame window.
SV_CYFULLSCREEN	Specifies the number of pels along the *y*-axis of the client window of a maximized frame window.
SV_CXICON	Specifies the number of pels along an icon's *x*-axis.
SV_CYICON	Specifies the number of pels along an icon's *y*-axis.
SV_CXPOINTER	Specifies the number of pels along the mouse pointer's *x*-axis.
SV_CYPOINTER	Specifies the number of pels along the mouse pointer's *y*-axis.
SV_DEBUG	Reserved.
SV_CURSORLEVEL	Specifies the cursor display count. The cursor is visible only when the display count is zero.
SV_POINTERLEVEL	Specifies the mouse-pointer display count. The mouse is visible only when the display count is zero.
SV_TRACKRECTLEVEL	Specifies the tracking-rectangle display count. The tracking rectangle is visible only when the display count is zero.
SV_CTIMERS	Specifies the number of available timers.
SV_CXBYTEALIGN	Specifies a horizontal alignment that is more efficient for the device driver.

Value	Meaning
SV_CYBYTEALIGN	Specifies a vertical alignment that is more efficient for the device driver.
SV_EXTRAKEYBEEP	Specifies whether beep is turned on for extended keys (keys not on an IBM PS/2 or compatible keyboard).
SV_SETLIGHTS	Specifies if the system controls the keyboard indicator lights.
SV_INSERTMODE	Specifies if insert mode is on or off for entry-field controls.
SV_MENUROLLDOWNDELAY	Specifies the delay for menu roll down.
SV_MENUROLLUPDELAY	Specifies the delay for menu roll up.
SV_ALTMNEMONIC	Specifies if the Alt key is allowed as a mnemonic.
SV_TASKLISTMOUSEACCESS	Specifies if the Task List can be accessed by the right mouse button.
SV_CSYSVALUES	Specifies the number of system values.

See Also **WinQuerySysValue**

Changes The following system values have been added:

SV_EXTRAKEYBEEP
SV_SETLIGHTS
SV_INSERTMODE
SV_MENUROLLDOWNDELAY
SV_MENUROLLUPDELAY
SV_ALTMNEMONIC
SV_TASKLISTMOUSEACCESS

■ WinSetWindowPos — Correction

```
BOOL WinSetWindowPos( hwnd, hwndInsertBehind, x, y, cx, cy, fs)
HWND hwnd;                  /* handle of window being set */
HWND hwndInsertBehind;      /* placement-order handle     */
SHORT x;                    /* horizontal position        */
SHORT y;                    /* vertical position          */
SHORT cx;                   /* width                      */
SHORT cy;                   /* height                     */
USHORT fs;                  /* window-positioning flags   */
```

The **WinSetWindowPos** function sets the position of a window.

Parameters

hwnd Identifies the window being set.

hwndInsertBehind Identifies relative window-placement order. This parameter is ignored if the *fs* parameter is not set to SWP_ZORDER. If this parameter is HWND_BOTTOM, the *hwnd* window is placed behind all sibling windows. If it is HWND_TOP, the *hwnd* window is placed on top of all sibling windows. Other values identify the sibling window behind which the *hwnd* window is placed.

x Specifies the horizontal position of the *hwnd* window (in window coordinates relative to the lower-left corner of its parent window). This parameter is ignored if the *fs* parameter is not set to SWP_MOVE.

y Specifies the vertical position of the *hwnd* window (in window coordinates relative to the lower-left corner of its parent window). This parameter is ignored if the *fs* parameter is not set to SWP_MOVE.

cx Specifies the horizontal window size (in device units). This parameter is ignored if the *fs* parameter is not set to SWP_SIZE.

cy Specifies the vertical window size (in device units). This parameter is ignored if the *fs* parameter is not set to SWP_SIZE.

fs Identifies the window-positioning options. This parameter can be one or more of the following values:

Value	Meaning
SWP_ACTIVATE	The window is activated and the focus to be set to the window that lost the focus the last time the frame window was deactivated. The activated window may not become the top window if it owns other frame windows.
SWP_DEACTIVATE	Deactivates the window, if it is the active window.
SWP_EXTSTATECHANGE	This flag is for application use. It is used to pass an additional flag to the portion of code that is handling messages.
SWP_FOCUSACTIVATE	Specifies that a frame window is receiving the focus. This flag is set so that an application that is processing the WM_ADJUSTWINDOWPOS message can tell if the message was sent as the result of a focus change.
SWP_FOCUSDEACTIVATE	Specifies that a frame window is losing the focus.
SWP_HIDE	Specifies that the window is to be hidden when created.
SWP_MAXIMIZE	With SWP_MINIMIZE, causes a window to be minimized, maximized, or restored. SWP_MAXIMIZE and SWP_MINIMIZE are mutually exclusive. If either SWP_MINIMIZE or SWP_MAXIMIZE is specified, then both SWP_MOVE and SWP_SIZE must also be specified. **WinSetWindowPos** and **WinSetMultWindowPos** depend on the previous state of the window; these flags cause the appropriate state to be toggled, as follows: the *x*, *y*, *cx*, and *cy* parameters specify the size and position to which the window will be restored if it is subsequently restored. This should be the normal size of the window.

Value	Meaning
SWP_MINIMIZE	See SWP_MAXIMIZE.
SWP_MOVE	Changes the window's *x,y* position.
SWP_NOADJUST	Does not send a WM_ADJUSTWINDOWPOS message to the window while processing (the window cannot readjust itself).
SWP_NOREDRAW	Does not redraw changes.
SWP_RESTORE	Restores a minimized or maximized window.
SWP_SHOW	Specifies that the window is to be shown when created.
SWP_SIZE	Changes the window size.
SWP_ZORDER	Changes the relative window placement.

Return Value

The return value is TRUE if the function is successful or FALSE if an error occurs.

Comments

If a window created with the CS_SAVEBITS style is moved, reduced in size, or hidden, the saved screen image is used to redraw the area uncovered when the window size changes, if those bits are still valid.

If the CS_SIZEREDRAW style is present, the entire window area is assumed invalid if sized. Otherwise, a WM_CALCVALIDRECTS message is sent to the window to inform the window manager which bits it is possible to preserve.

Messages sent from **WinSetWindowPos** and **WinSetMultWindowPos** have specific orders within the window positioning process. The process begins with redundancy checks and precalculations on every window for each requested operation. For example, if SWP_SHOW is present but the window is already visible, then SWP_SHOW is turned off. If SWP_SIZE is present and the new size is equal to the previous size, SWP_SIZE is turned off. If the operations will create new results, the information is calculated and stored. For example, if being sized or moved, the new window rectangle is stored for later use. At this point, the WM_ADJUSTWINDOWPOS message is sent to any window that is being sized or moved. Also at this point, the WM_CALCVALIDRECTS message is sent to any window that is being sized and that does not have the CS_SIZEREDRAW window style.

When the new window state is calculated, the window-management process begins. Window areas that can be preserved are moved from the old to the new positions, window areas that are invalidated by these operations are calculated and distributed as update regions, and so forth. When this is finished, and before any synchronous-paint windows are repainted, the WM_SIZE message is sent to any windows that have changed size. Next, all the synchronous-paint windows that can be repainted are repainted and the entire process is complete.

If a synchronous-paint parent window has a size-sensitive area displayed that includes synchronous-paint child windows, the parent window will reposition those windows when it receives the WM_SIZE message. Their invalid regions will be added to the parent window's invalid region, resulting in one update after the parent window's WM_SIZE message, rather than many independent and subsequently duplicated updates.

Certain windows are not positioned precisely to the parameters specified by this function. For example, frame windows without the FCF_NOBYTEALIGN style creation flag are not positioned to any specific screen coordinate.

The following messages are sent by this function:

Value	Meaning
WM_CALCVALIDRECTS	Sent to determine the area of a window that it may be possible to preserve as the window is sized.
WM_SIZE	Sent if the size of the window has changed, after the change has been effected.
WM_MOVE	Sent when a window with CS_MOVENOTIFY class style moves its absolute position.
WM_ACTIVATE	Sent if a different window becomes the active window. For more information, see the **WinSetActiveWindow** function.
WM_ADJUSTWINDOWPOS	Sent if SWP_NOADJUST is not specified. The message's *mp1* parameter points to an SWP structure that has been filled in by the **WinSetWindowPos** function with the proposed move/size data. The window can adjust this new position by changing the contents of the SWP structure.

Example

This example gets the dimensions of the desktop window, and calls **WinSetWindowPos** to place the application's frame window in the upper left corner. By positioning the window relative to the desktop window, the window position is device-independent; it will work on any display adapter no matter what the vertical and horizontal resolution is.

```
RECTL rcl;

WinQueryWindowRect(HWND_DESKTOP, &rcl);
WinSetWindowPos(hwndFrame, HWND_TOP,
    rcl.xLeft,                                          /* x pos  */
    rcl.yTop - 60,                                      /* y pos  */
    140,                                                /* x size */
    60,                                                 /* y size */
    SWP_ACTIVATE | SWP_MOVE | SWP_SIZE | SWP_SHOW);     /* flags  */
```

See Also

WinSetActiveWindow, WinSetMultWindowPos, WM_ADJUSTWINDOWPOS, WM_CALCVALIDRECTS

Corrections

Certain windows are not positioned precisely to the parameters specified by this function. For example, frame windows without a style creation flag of FCF_NOBYTEALIGN are not positioned to any specific screen coordinate.

■ WinSwitchToProgram — New

USHORT WinSwitchToProgram(*hSwitch* **)**
HSWITCH *hSwitch*; /* handle of application to activate */

The **WinSwitchToProgram** function makes an application the active application. The function succeeds only if the calling application is currently the active application (the application with the active window).

Parameters *hSwitch* Identifies the application to make active.

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

PMERR_INVALID_SWITCH_HANDLE

See Also **WinInstStartApp**

■ WinTerminateApp — New

```
BOOL WinTerminateApp( happ)
HAPP happ;
```

The **WinTerminateApp** function terminates an application previously started with the **WinInstStartApp** function.

Parameters *happ* Identifies the application to terminate.

Return Value The return value is TRUE if the application is terminated successfully or NULL if an error occurs.

Errors Use the **WinGetLastError** function to retrieve the error value, which may be one of the following:

PMERR_INVALID_HAPP
PMERR_CANNOT_STOP

Comments The application to terminate must have been started using the **WinInstStartApp** function with the SAF_STARTCHILDAPP option specified.

If the specified application does not stop, this function returns TRUE. To ensure that the application has terminated, the application calling **Win-TerminateApp** must wait for the appropriate message to be posted to the window specified in the **WinInstStartApp** function.

See Also **WinInstStartApp**

■ WinWindowFromID — Correction

```
HWND WinWindowFromID( hwndParent, id)
HWND hwndParent;    /* parent-window handle */
USHORT id;          /* window identifier    */
```

The **WinWindowFromID** function returns the first child window that has the specified identifier of the specified parent window.

Parameters *hwndParent* Identifies the parent window.

id Identifies the window.

Return Value The return value is a window handle. If no child window exists with identifier *id* the return value is NULL.

Comments To obtain the window handle for an item within a dialog box, the *hwndParent* parameter is set to the dialog-box window's handle and the *id* parameter is set to the identifier of the item in the dialog template.

To obtain the window handle for a frame control, the *hwndParent* parameter is set to the frame window's handle and the *id* parameter is set to one of the FID constants, indicating which frame control you want a handle of.

The following list contains the frame control identifiers. Note that you must also define the INCL_WINFRAMEMGR constant before including *pmwin.h*

Value	Meaning
FID_CLIENT	Identifies the client window.
FID_HORZSCROLL	Identifies the horizontal scroll bar.
FID_MENU	Identifies the application menu.
FID_MINMAX	Identifies the minimize/maximize box.
FID_SYSMENU	Identifies the system menu.
FID_TITLEBAR	Identifies the title bar.
FID_VERTSCROLL	Identifies the vertical scroll bar.

Example

This example calls **WinWindowFromID** to get the window handle of the system menu and calls **WinSendMsg** to send a message to disable the Close menu item.

```
#define INCL_WINMESSAGEMGR    /* includes message manager functions */
#define INCL_WINFRAMEMGR      /* includes FID_ constants            */
#include <os2.h>

HWND hwndSysMenu;

hwndSysMenu = WinWindowFromID(hwndDlg, FID_SYSMENU);
WinSendMsg(hwndSysMenu, MM_SETITEMATTR,
    MPFROM2SHORT(SC_CLOSE, TRUE),
    MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
```

See Also

WinMultWindowFromIDs, WinWindowFromPoint

Corrections

The list of FID constants incorrectly identified FID_MENU as referring to the system menu. It actually refers to the application menu.

■ WinWindowFromPoint — Change

```
HWND WinWindowFromPoint(hwnd, pptl, fChildren, fLock)
HWND hwnd;          /* handle of the window                    */
PPOINTL pptl;       /* address of structure with the point     */
BOOL fChildren;     /* scope flag                              */
BOOL fLock;         /* lock/unlock flag                        */
```

The **WinWindowFromPoint** function finds the window that is below a specified point and that is a descendant of a specified window. This function checks only the descendants of the specified window.

Parameters

hwnd Identifies the window whose child windows are tested.

pptl Points to a **POINTL** structure that contains the point to test, specified in window coordinates relative to the *hwnd* parameter. The **POINTL** structure has the following form:

```
typedef struct _POINTL  {
    LONG  x;
    LONG  y;
} POINTL;
```

fChildren Specifies which child windows to test. If *fChildren* is TRUE, the function tests all the descendants of *hwnd*, including child windows of child windows. If *fChildren* is FALSE, the function tests only the immediate child windows of *hwnd*.

fLock This parameter is ignored by MS OS/2 1.2 and later versions.

Return Value If *fChildren* is FALSE, the return value is *hwnd*, a child of *hwnd*, or NULL. If *fChildren* is TRUE, the return value is the topmost window if that window is *hwnd* or a child of *hwnd*—unless another window of CS_HITTEST type is found, in which case the window returned may not be the topmost window.

See Also **WinWindowFromID**

Changes The *fLock* parameter is ignored by MS OS/2 1.2 and later versions.

■ WinWriteProfileData Change

```
BOOL WinWriteProfileData( hab, pszAppName, pszKeyName, pchBinaryData, cchData )
HAB hab;                 /* handle of anchor block      */
PSZ pszAppName;          /* address of application name */
PSZ pszKeyName;          /* address of keyname          */
PVOID pchBinaryData;     /* address of data             */
USHORT cchData;          /* length of data              */
```

The **WinWriteProfileData** function places binary data into the *os2.ini* file. The placement of the data is determined by an application name and a keyname that are passed to the function. The data can subsequently be retrieved by using the **WinQueryProfileData** function, specifying the same application name and keyname as are given in the *pszAppName* and *pszKeyName* parameters.

Parameters *hab* Identifies the anchor block.

pszAppName Points to a null-terminated string that contains the name of the application. The length of the string must be less than 1024 bytes, including the null terminating character. The application name is case-sensitive. If there is no application field in the *os2.ini* file that matches *pszAppName*, a new application field is created.

pszKeyName Points to a null-terminated string that contains the keyname. The length of the string must be less than 1024 bytes, including the null terminating character. If *pszKeyName* is NULL, all keynames and their data are deleted. The keyname is case-sensitive. If there is no keyname that matches *pszKeyName*, a new keyname field is created. If the keyname already exists, the existing value is overwritten.

pchBinaryData Points to the binary data that is placed into the *os2.ini* file. There is no explicit termination character. If *pchBinaryData* is NULL, the previous value associated with the *pszKeyName* parameter is deleted; otherwise, the data string becomes the value, even if it has a zero length. The amount of data should not exceed 64K.

cchData Specifies the size (in bytes) of the *pchBinaryData* parameter.

Return Value The return value is TRUE if the function is successful, or FALSE if an error occurs. If the *os2.ini* file exists but is in corrupted form, **WinWriteProfileData** returns FALSE.

Comments The **WinWriteProfileData** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfWriteProfileData** function.

See Also **PrfWriteProfileData, WinQueryProfileData**

Changes This function has been replaced by the **PrfWriteProfileData** function.

■ WinWriteProfileString — Change

```
BOOL WinWriteProfileString( hab, pszAppName, pszKeyName, pszString)
HAB hab;             /* handle of anchor block        */
PSZ pszAppName;      /* address of application name   */
PSZ pszKeyName;      /* address of keyname            */
PSZ pszString;       /* address of string to write    */
```

The **WinWriteProfileString** function places an ASCII string into the *os2.ini* file. The placement of the string is determined by an application name and a keyname that are passed to the function. The string can subsequently be retrieved by using the **WinQueryProfileString** function, specifying the same application name and keyname as are given in the *pszAppName* and *pszKeyName* parameters.

Parameters

hab Identifies the anchor block.

pszAppName Points to a null-terminated string that contains the name of the application. The length of the string must be less than 1024 bytes, including the null terminating character. The application name is case-sensitive. If there is no application field in the *os2.ini* file that matches *pszAppName*, a new application field is created.

pszKeyName Points to a null-terminated text string that contains the keyname. The length of the string must be less than 1024 bytes, including the null terminating character. If *pszKeyName* is NULL, all keynames and their data are deleted. The keyname is case-sensitive. If there is no keyname that matches *pszKeyName*, a new keyname field is created. If the keyname already exists, the existing value is overwritten.

pszString Points to a null-terminated ASCII string that is placed into the *os2.ini* file. If *pszString* is NULL, the previous value associated with *pszKeyName* is deleted; otherwise, the ASCII string becomes the value, even if it has a zero length. The size of the string should not exceed 64K.

Return Value The return value is TRUE if the function is successful, or FALSE if an error occurs. Use the **WinGetErrorInfo** function to retrieve the error value, which may be one of the following:

PMERR_CAN_NOT_CALL_SPOOLER
PMERR_INVALID_PARM

Comments The **WinWriteProfileString** function provides compatibility with MS OS/2 1.1 and earlier versions. Applications intended exclusively for MS OS/2 1.2 and later versions should use the **PrfWriteProfileString** function.

See Also **PrfWriteProfileString, WinQueryProfileString**

Changes This function has been replaced by the **PrfWriteProfileString** function.

■ WM_ADJUSTWINDOWPOS Change

```
WM_ADJUSTWINDOWPOS
pswp = (PSWP) PVOIDFROMMP(mp1);    /* pointer to SWP structure */
```

The WM_ADJUSTWINDOWPOS message is sent when a window is about to be moved or sized. It gives the window an opportunity to adjust the new size and position before the window is actually moved and sized.

Parameters *pswp* Low and high word of *mp1*. Points to an SWP structure that contains the new window size and position information. The **SWP** structure has the following form:

```
typedef struct _SWP {
    USHORT fs;
    SHORT  cy;
    SHORT  cx;
    SHORT  y;
    SHORT  x;
    HWND   hwndInsertBehind;
    HWND   hwnd;
} SWP;
```

Return Value An application should return FALSE if it does not change the SWP structure. Otherwise, it should return on of the following values:

Value	Meaning
AWP_MINIMIZED	The window was minimized.
AWP_MAXIMIZED	The window was maximized.
AWP_RESTORED	The window was restored.
AWP_ACTIVATE	The window was activated.
AWP_DEACTIVATE	The window was deactivated.

See Also **WinCreateWindow**, WM_CALCVALIDRECTS, WM_WINDOWPOSCHANGED

Changes An application should return FALSE if it does not change the **SWP** structure. Otherwise, it should return on of the following values:

Value	Meaning
AWP_MINIMIZED	The window was minimized.
AWP_MAXIMIZED	The window was maximized.
AWP_RESTORED	The window was restored.

Value	Meaning
AWP_ACTIVATE	The window was activated.
AWP_DEACTIVATE	The window was deactivated.

■ WM_APPTERMINATENOTIFY New

```
WM_APPTERMINATENOTIFY
mp1 = MPFROMLONG((HAPP) happ);           /* application handle */
mp2 = MPFROMSHORT((USHORT) usRetCode);   /* return code        */
```

The WM_APPTERMINATENOTIFY message is sent when a child application started by the **WinInstStartApp** function terminates.

Parameters

happ Low word of *mp1*. Identifies the application returned by the **WinInstStartApp** function.

usRetCode Low word of *mp2*. Specifies the return code from the application that has terminated.

Return Value An application should return zero if it processes this message.

See Also **WinInstStartApp, WinTerminateApp**

■ WM_CALCFRAMERECT New

```
WM_CALCFRAMERECT
prclFrame = (PRECTL) PVOIDFROMMP(mp1); /* pointer to RECTL structure */
fClient = (BOOL) SHORT1FROMMP(mp2);    /* client-indicator flag      */
```

The WM_CALCFRAMERECT message is sent to a frame window when the **WinCalcFrameRect** function is called. The default window procedure calculates a client rectangle from a frame rectangle or calculates a frame rectangle from a client rectangle.

Parameters

prcl Low word of *mp1*. Points to the **RECTL** structure that contains the coordinates of the window. If the *fClient* parameter is TRUE, this structure contains the coordinates of the frame window, and on return, it contains the coordinates of a client window. If the *fClient* parameter is FALSE, this structure contains the coordinates of the client window, and on return, it contains the coordinates of a frame window.

The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

fClient Low word of *mp2*. Specifies whether the window to calculate is a client window or a frame window. If this value is TRUE, a client window is calculated. If this value is FALSE, a frame window is calculated.

Return Value

If an application processes this message, it should return TRUE if successful or FALSE if an error occurs or the calculated rectangle is empty.

See Also

WinCalcFrameRect

■ WM_CALCVALIDRECTS — Correction

```
WM_CALCVALIDRECTS
parclWindow = (PRECTL) PVOIDFROMMP(mp1);    /* source rectangle   */
pswpDest = (PSWP) PVOIDFROMMP(mp2);         /* destination window */
```

The WM_CALCVALIDRECTS message is sent when a window is about to be resized. This allows the application to specify the coordinates of a rectangle that will be preserved and to designate where this rectangle will be moved in the resized window. Areas outside this rectangle will be redrawn.

Parameters

parclWindow Low and high word of *mp1*. Points to an array of two **RECTL** structures that contain the dimensions of the window before and after resizing. The first **RECTL** structure contains the source rectangle; the second **RECTL** structure contains the destination rectangle. The coordinates of the rectangles are relative to the parent window of the window. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

pswpDest Low and high word of *mp2*. Points to the **SWP** structure that contains information about the window after it is resized. The **SWP** structure has the following form:

```
typedef struct _SWP {
    USHORT fs;
    SHORT  cy;
    SHORT  cx;
    SHORT  y;
    SHORT  x;
    HWND   hwndInsertBehind;
    HWND   hwnd;
} SWP;
```

Return Value

If an application processes this message, it can return zero to indicate it has changed the rectangle itself, CVR_REDRAW if the entire window is to be redrawn, or a combination of the following values:

Value	Meaning
CVR_ALIGNBOTTOM	Align with the bottom edge of the window.
CVR_ALIGNLEFT	Align with the left edge of the window.
CVR_ALIGNRIGHT	Align with the right edge of the window.
CVR_ALIGNTOP	Align with the top edge of the window.

Comments The WM_CALCVALIDRECTS message is not sent if a window has the CS_SIZEREDRAW style because such windows are always completely redrawn when resized.

See Also WM_ADJUSTWINDOWPOS

Corrections The first parameter points to an array of two **RECTL** structures. The first structure is the source rectangle; the second structure is the destination rectangle.

The second parameter points to an **SWP** structure, not to a **RECTL** structure.

The CVR_REDRAW was incorrectly spelled CV_REDRAW.

■ WM_CHAR — Change

```
WM_CHAR
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);     /* key flags              */
uchRepeat = (UCHAR) CHAR3FROMMP(mp1);        /* repeat count           */
uchScanCode = (UCHAR) CHAR4FROMMP(mp1);      /* scan code              */
usChr1 = (UCHAR) CHAR1FROMMP(mp2);           /* character              */
usChr2 = (UCHAR) CHAR2FROMMP(mp2);           /* 2nd byte of character */
usVKey = (USHORT) SHORT2FROMMP(mp2);         /* virtual key            */
```

The WM_CHAR message is sent whenever the user presses a key. This message is placed in the queue associated with the window that has the focus.

Parameters *fsKeyFlags* Low word of *mp1*. Specifies the keyboard control codes. It can be one or more of the following values:

Value	Meaning
KC_CHAR	The *usChr* parameter value is valid; otherwise, *mp2* contains zero.
KC_SCANCODE	The *uchScanCode* parameter value is valid; otherwise, *uchScanCode* contains zero.
KC_VIRTUALKEY	The *usVKey* parameter value is valid; otherwise, *usVKey* contains zero.
KC_KEYUP	The event was a key-up transition; otherwise, it was a key-down transition.
KC_PREVDOWN	The key was previously down; otherwise, it was previously up.
KC_DEADKEY	The character code is a dead key. The application must display the glyph for the dead key without advancing the cursor.
KC_COMPOSITE	The character code was formed by combining the current key with the previous dead key.
KC_INVALIDCOMP	The character code was not a valid combination with the preceding dead key. The application must advance the cursor past the dead-key glyph and then, if the current character is *not* a space, it must beep the speaker and display the new character code.
KC_LONEKEY	This bit is set if the key was pressed and released without any other keys being pressed or released between the time the key was pressed and released.

Value	Meaning
KC_SHIFT	The shift state was active when the key is pressed or released.
KC_ALT	The ALT state was active when the key was pressed or released.
KC_CTRL	The CONTROL state was active when the key was pressed or released.

uchRepeat Low byte of high word of *mp1*. Specifies the repeat count of the key.

uchScanCode High byte of high word of *mp1*. Specifies the character scan code of the character. *usChr1* First byte of the low word of *mp2*. Specifies the ASCII character.

usChr2 Second byte of the low word of *mp2*, for double-byte characters only. Specifies second byte of the character, or is zero for standard ASCII.

usVKey High word of *mp2*. Specifies the virtual-key code.

Comments

Generally, all WM_CHAR messages generated from actual user input have the KC_SCANCODE code set. However, if the message has been generated by an application that has issued the **WinSetHook** function to filter keystrokes, or if it was posted to the application queue, this code may not be set.

The **CHARMSG** macro can be used to access the WM_CHAR message parameters. This macro defines a **CHARMSG** structure pointer that has the following form:

```
struct _CHARMSG {
    USHORT chr;              /* mp2 */
    USHORT vkey;
    USHORT fs;               /* mp1 */
    UCHAR  cRepeat;
    UCHAR  scancode;
};
```

When the character returned is a double-byte character, then the second byte of *mp2* contains the second byte of the character. For standard ASCII, the second byte is zero.

Example

This example uses the **CHARMSG** macro to process a WM_CHAR message. It first uses the macro to determine if a key was released. It then uses the macro to generate a **switch** statement based on the character received.

```
MRESULT CALLBACK GenericWndProc(hwnd, usMessage, mp1, mp2)
HWND   hwnd;
USHORT usMessage;
MPARAM mp1;
MPARAM mp2;
{

    switch (usMessage) {
    case WM_CHAR:
        if (CHARMSG(&usMessage)->fs & KC_KEYUP) {
            switch (CHARMSG(&usMessage)->chr) {
```

Return Value

An application should return TRUE if it processes the message; otherwise it should return FALSE.

See Also

WinSetHook, WM_NULL, WM_TRANSLATEACCEL, WM_VIOCHAR

Changes For double-byte character sets, the second parameter (*mp2*) of WM_CHAR contains both bytes of the double-byte character.

■ WM_CLOSE — Change

```
WM_CLOSE
```

The WM_CLOSE message is sent as a signal that the window or its application should terminate. This message allows the window to control the termination process.

Parameters This message does not use any parameters.

Return Value An application should return zero if it processes this message.

Comments If WM_CLOSE is passed to the **WinDefDlgProc** function, the function calls the **WinDismissDlg** function and passes the DID_CANCEL result code to it.

Example In the following example, the *fChanges* variable is checked. If it is TRUE, the user is asked if he or she wants to exit without saving any changes. If the user responds by choosing the No button, then zero is returned and the application does not exit. If the user responds by choosing the Yes button, then a WM_QUIT message is posted so that the application will terminate.

```
case WM_CLOSE:
    if (fChanges) {
        if (WinMessageBox(HWND_DESKTOP, hwndClient,
                "Do you want to exit without saving your changes?",
                "", 0, MB_NOICON | MB_YESNO) == MBID_NO)
            return (0L);
    }
    WinPostMsg(hwnd, WM_QUIT, 0L, 0L);
    return (0L);
```

See Also **WinDefWindowProc, WinMessageBox, WinPostMsg,** WM_QUIT

Changes If a dialog window has a system menu, selecting the "Close" menu item calls the **WinDismissDlg** function, passing the DID_CANCEL result code. Previous versions of MS OS/2 closed the application, rather than only the dialog box.

■ WM_DRAWITEM — Correction

```
WM_DRAWITEM
id = (USHORT) SHORT1FROMMP(mp1);        /* window ID             */
poi = (POWNERITEM) PVOIDFROMMP(mp2);    /* pointer to OWNERITEM  */
```

The WM_DRAWITEM message is sent to the owner of a list box when an item in an owner-drawn list needs to be drawn or highlighted. The list box must have the LS_OWNERDRAW style. The WM_DRAWITEM message is also sent to the owner of a menu when an item in the owner-drawn menu needs to be drawn or highlighted. The menu must have the MIS_OWNERDRAW style.

Parameters *id* Low word of *mp1*. Identifies the window of the list-box or menu control sending this message.

poi Low and high word of *mp2*. Points to an **OWNERITEM** structure. The **OWNERITEM** structure has the following form:

```
typedef struct _OWNERITEM {
    HWND    hwnd;
    HPS     hps;
    USHORT  fsState;
    USHORT  fsAttribute;
    USHORT  fsStateOld;
    USHORT  fsAttributeOld;
    RECTL   rclItem;
    SHORT   idItem;
    ULONG   hItem;
} OWNERITEM;
```

Return Value

The application should return TRUE if it draws the list-box item; it should return FALSE if the list box should draw the item. If the WM_DRAWITEM message is sent to a menu, the return value is ignored.

Comments

When an item is to be drawn, the **fsState** field and the **fsStateOld** field of the **OWNERITEM** structure will be equal. The application should draw the item and return TRUE, or it should return FALSE to let the list box draw the item. The list box can draw only text items, so the application must handle the drawing of other types of objects.

When an item is to be highlighted, the **fsState** field is TRUE and the **fsStateOld** field is FALSE. In this case, the application should carry out the highlighting and set **fsState** and **fsStateOld** equal to FALSE before returning TRUE, or it should return FALSE so the list box can perform default highlighting of the item.

When highlighting is to be removed from an item, the **fsState** field is FALSE and the **fsStateOld** field is TRUE. An application can remove the highlighting, set the **fsState** and **fsStateOld** equal to FALSE and return TRUE, or it can return FALSE to let the list box remove the highlighting.

See Also

LM_QUERYITEMTEXT

Corrections

The application should return TRUE if it draws the list-box item; it should return FALSE if the list box should draw the item. If the WM_DRAWITEM message is sent to a menu, the return value is ignored.

■ WM_FORMATFRAME — Correction

```
WM_FORMATFRAME
paswp = (paswp) PVOIDFROMMP(mp1);   /* pointer to SWP array      */
prcl = (PRECTL) PVOIDFROMMP(mp2);   /* pointer to RECTL structure */
```

The WM_FORMATFRAME message is sent to a frame window to calculate the sizes and positions of the frame controls and the client window. The frame-window procedure sends the message to its client window and, if the client window returns TRUE (indicating that it processed the message), no further action occurs. Otherwise, the frame window calls the **WinFormatFrame** function.

Parameters

paswp Low and high word of *mp1*. Points to an array of **SWP** structures. The array elements are filled in the order of the FID values of the frame controls, with the FID_CLIENT window always the last element in the array. The **SWP** structure has the following form:

```
typedef struct _SWP {
    USHORT fs;
    SHORT  cy;
    SHORT  cx;
    SHORT  y;
    SHORT  x;
    HWND   hwndInsertBehind;
    HWND   hwnd;
} SWP;
```

prcl Low and high word of *mp2*. Points to a **RECTL** structure that contains the rectangle within which the frame controls are formatted. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

Return Value An application should return TRUE if it processes this message.

Comments Note that the *paswp* parameter points to memory allocated according to the value returned by the WM_QUERYFRAMECTLCOUNT message. The application must not write beyond this area.

See Also **WinFormatFrame**

Corrections The parameters were reversed. The first parameter is the array of **SWP** structures; the second parameter is a pointer to a **RECTL** structure.

■ WM_MEASUREITEM Change

```
WM_MEASUREITEM
id = SHORT1FROMMP(mp1);                      /* list-box identifier  */
poi = (POWNERITEM) PVOIDFROMMP(mp2);         /* pointer to OWNERITEM */
```

The WM_MEASUREITEM message is sent to calculate the height of each item in a window. It is normally sent to list boxes and menus. All items are the same height in a list box or menu.

Parameters *id* Low word of *mp1*. Specifies the window.

poi Low and high word of *mp2*. When this message is sent to a menu window, this parameter points to an **OWNERITEM** structure. Otherwise, this parameter is not used. The **OWNERITEM** structure has the following form:

```
typedef struct _OWNERITEM {
    HWND    hwnd;
    HPS     hps;
    USHORT  fsState;
    USHORT  fsAttribute;
    USHORT  fsStateOld;
    USHORT  fsAttributeOld;
    RECTL   rclItem;
    SHORT   idItem;
    ULONG   hItem;
} OWNERITEM;
```

Return Value If this message is processed by a list box, the low word of the return value contains the height of the list-box item. If the style LS_HORZSCROLL is set, the high word contains the length of the list-box item; otherwise, the high word must be set to zero.

If this message is processed by a menu, the return value is ignored. The width and height are returned by placing their dimensions in the **OWNERITEM** structure passed in the *poi* parameter.

See Also LM_SETITEMHEIGHT

Changes If the style LS_HORZSCROLL is set, WM_MEASUREITEM must return the length of the list-box item as the high word of the return value.

■ WM_MOVE — Correction

```
WM_MOVE
```

The WM_MOVE message is sent when a window with CS_MOVENOTIFY style changes its absolute position or when a parent window of that window is moved. The window's new position can be obtained by calling the **WinQueryWindowPos** function.

Parameters This message does not use any parameters.

Return Value An application should return zero if it processes this message.

See Also **WinQueryWindowPos**

Corrections Use the **WinQueryWindowPos** function, not the **WinQueryWindowRect** function to obtain the window's position.

■ WM_PRESPARAMCHANGED — New

```
WM_PRESPARAMCHANGED
idParam = (ULONG) LONGFROMMP(mp1);    /* presentation-parameter ID */
```

The WM_PRESPARAMCHANGED message is sent when a presentation parameter has changed.

Parameters *idParam* Low and high word of *mp1*. Identifies the presentation parameter that changed. This parameter can be one of the following values:

Value	Meaning
PP_FOREGROUNDCOLOR	RGB foreground color
PP_FOREGROUNDCOLORINDEX	Color index of foreground color
PP_BACKGROUNDCOLOR	RGB background color

Value	Meaning
PP_BACKGROUNDCOLORINDEX	Color index of background color
PP_HILITEFOREGROUNDCOLOR	RGB color of foreground highlighted area
PP_HILITEFOREGROUNDCOLORINDEX	Color index of foreground highlighted area
PP_HILITEBACKGROUNDCOLOR	RGB color of background highlighted area
PP_HILITEBACKGROUNDCOLORINDEX	Color index of background highlighted area
PP_DISABLEDFOREGROUNDCOLOR	RGB foreground disabled color
PP_DISABLEDFOREGROUNDCOLORINDEX	Color index of foreground disabled color
PP_DISABLEDBACKGROUNDCOLOR	RGB color of background disabled color
PP_DISABLEDBACKGROUNDCOLORINDEX	Color index of background disabled color
PP_BORDERCOLOR	RGB color of window border
PP_BORDERCOLORINDEX	Color index of window border
PP_FONTNAMESIZE	Font size
PP_FONTHANDLE	Font handle

See Also **WinQueryPresParam, WinSetPresParam**

■ WM_QUERYHELPINFO New

```
WM_QUERYHELPINFO
```

The WM_QUERYHELPINFO message is sent to a frame window to retrieve the handle of the help instance.

Parameters This message does not use any parameters.

Return Value An application should return the help instance handle associated with the window. If no handle is available, the application should return NULL.

See Also WM_SETHELPINFO

■ WM_SAVEAPPLICATION New

```
WM_SAVEAPPLICATION
mp1 = 0L;    /* not used, must be zero */
mp2 = 0L;    /* not used, must be zero */
```

The WM_SAVEAPPLICATION message notifies an application to save its current state (for example, due to a pending system shutdown).

Parameters This message does not use any parameters.

Comments When a system shutdown is requested, MS OS/2 enumerates the applications in the Task List and sends each application a WM_SAVEAPPLICATION message. The sender of the WM_SAVEAPPLICATION message suspends execution until it receives a reply. The receiving application must not display dialog or message boxes. Doing so could delay the reply and result in unacceptable delays in completing the shutdown.

In MS OS/2, version 1.2, the application must save its state to the *os2.ini* file by using the **WinWriteProfileString** or **WinWriteProfileData** function, or it must save its state to some other file.

To be compatible with future releases of MS OS/2, an application should call **WinDefWindowProc** after processing the WM_SAVEAPPLICATION message.

Each application should maintain only one "saved state." If an application receives multiple WM_SAVEAPPLICATION messages, it should overwrite the previous "saved state" with a new "saved state" for each new WM_SAVEAPPLICATION message.

See Also **WinDefWindowProc, WinWriteProfileData, WinWriteProfileString**

■ WM_SETHELPINFO New

```
WM_SETHELPINFO
hwnd = HWNDFROMMP(mp1);    /* handle of help table */
```

The WM_SETHELPINFO message is sent to a frame window to set the handle of the help instance for that window.

Parameters *hwnd* Low and high word of *mp1*. Identifies the help instance.

Return Value An application should return zero if it processes this message.

See Also WM_QUERYHELPINFO

■ WM_WINDOWPOSCHANGED New

```
WM_WINDOWPOSCHANGED
mp1 = MPFROMP(<paswp>);          /* pointer to array of SWP structures */
mp2 = MPFROMLONG(<flReturn>);    /* return-value flag                  */
```

The WM_WINDOWPOSCHANGED message is sent whenever the size of a window changes.

Parameters

paswp Low and high word of *mp1*. Points to an array of two **SWP** structures: the first **SWP** structure contains the new state of the window; the second **SWP** structure contains the previous state of the window. The **SWP** structure has the following form:

```
typedef struct _SWP {
    USHORT fs;
    SHORT  cy;
    SHORT  cx;
    SHORT  y;
    SHORT  x;
    HWND   hwndInsertBehind;
    HWND   hwnd;
} SWP;
```

flReturn Specifies the return value of the WM_ADJUSTWINDOWPOS message; it is FALSE if SWP_NOADJUST was specified.

Comments

The entire window state is filled in both **SWP** structures; however, the **fs** field of the first **SWP** structure contains only those bits that correspond to the actual changes that occurred. For example, if a window is resized, fields **x** and **y** contain the position of the window even though it did not move, but the **fs** field does not contain the SWP_MOVE flag.

Example

This example processes the WM_WINDOWPOSCHANGED message and assigns the two structures to pointers:

```
PSWP pswpNew, pswpOld;

case WM_WINDOWPOSCHANGED:
    pswpNew = PVOIDFROMMP(mp1);
    pswpOld = pswpNew + 1;
```

See Also

WinCreateWindow, WM_ADJUSTWINDOWPOS, WM_CALCVALIDRECTS

Chapter
4

Types, Macros, Structures

4.1 Introduction 361
4.2 Types 362
4.3 Macros 362
4.4 Structures 362

4.1 Introduction

This chapter describes the new and updated types, macros, and structures used with MS OS/2, version 1.2, functions and messages. For a complete list of all MS OS/2 types, macros, and structures, see the *Microsoft Operating System/2 Programmer's Reference, Volume* 2 and *Volume 3.*

The MS OS/2 functions use many types, macros, and structures that are not part of the standard C language. These types, macros, and structures have been defined to make the task of creating MS OS/2 programs easier and to make programs sources clearer and easier to understand.

All types, macros, and structures in this manual are defined in the MS OS/2 C-language include files. You may also want to use these when developing MS OS/2 programs in other computer languages, such as Pascal or assembly language. If include files for a given language are not available, you can translate the definitions given in this chapter by following these guidelines:

- Numbers must be integers or fixed-point real numbers. MS OS/2 functions do not support floating-point numbers. An MS OS/2 program can use floating-point numbers as long as an appropriate run-time library or coprocessor is supplied and floating-point numbers are not used as parameters to the MS OS/2 functions.
- Structures must be packed. Some compilers align each new field in a structure on word or double-word boundaries. This may leave unused bytes in a structure if a given field is smaller than the width between boundaries. MS OS/2 functions require that unused bytes be removed from structures.
- Reserved fields in structures should be set to zero. Unless otherwise specified, MS OS/2 functions expect reserved fields to be set to zero to avoid compatibility problems with future releases of MS OS/2.
- Variable-length structures must be supported. Several MS OS/2 functions use variable-length structures to receive and/or return information. In a variable-length structure, the number of fields in the structure varies depending on when the structure is used. In the C language, programs typically support variable-length structures by allocating enough memory for the current number of fields and accessing those fields by using a pointer to the structure. Programs in other languages may use this method or devise their own method for supporting variable-length structures.
- All 16-bit pointers must be relative to an explicitly defined segment register. Some compilers assume that the **ds** and **ss** registers contain the same value and implicitly use one segment for both. MS OS/2 does not guarantee that the **ds** and **ss** registers will be equal. This is especially true in dynamic-link libraries and programs that use callback functions (for example, window procedures).
- All 32-bit pointers must consist of a *selector:offset* pair. MS OS/2 functions do not use physical addresses (that is, an address that represents a 32-bit offset from the beginning of physical memory). (One exception to this rule is the **VioGetPhysBuf** function, which requires a physical address to video memory.)

4.2 Types

The following data types are new or modified for MS OS/2, version 1.2:

Type	Meaning
HAPP	32-bit value used as an application handle.
HINI	32-bit value used as an initialization-file handle.
HLIB	16-bit value used as a module handle.
HPROC	32-bit value used as a pointer to a procedure (function).
LINE	32-bit value used as a line number.
PHINI	32-bit value used as a pointer to an initialization-file handle.

4.3 Macros

There are no new or updated macros for MS OS/2, version 1.2.

4.4 Structures

The following structures are used by the MS OS/2, version 1.2, functions described in this manual.

■ AVAILDATA New

```
typedef struct _AVAILDATA {     /* avldt */
    USHORT  cbpipe;
    USHORT  cbmessage;
} AVAILDATA;
```

The **AVAILDATA** structure contains information about the bytes in a named pipe.

Fields

cbpipe Specifies the number of bytes left in the pipe.

cbmessage Specifies the number of bytes left in the current message.

See Also

DosPeekNmPipe

■ CHARBUNDLE Change

```
typedef struct _CHARBUNDLE {     /* cbnd */
    LONG    lColor;
    LONG    lBackColor;
    USHORT  usMixMode;
    USHORT  usBackMixMode;
    USHORT  usSet;
    USHORT  usPrecision;
    SIZEF   sizfxCell;
    POINTL  ptlAngle;
    POINTL  ptlShear;
    USHORT  usDirection;
} CHARBUNDLE;
```

The **CHARBUNDLE** structure contains fields that describe the current character attributes in the application's presentation space. MS OS/2 uses these attributes whenever the application draws text using one of the **Gpi** functions.

Fields

lColor Specifies the character foreground color.

lBackColor Specifies the character background color.

usMixMode Specifies the foreground mix mode. MS OS/2 uses this mix mode when it combines the character foreground color and the current drawing-surface color.

usBackMixMode Specifies the background mix mode. MS OS/2 uses this mix mode when it combines the character background color and the current drawing-surface color.

usSet Specifies the character set. This value is the local identifier for the current logical font. It can be any value from 1 through 254.

usPrecision Specifies the current character mode. There are three possible modes: mode 1, mode 2, and mode 3. If mode 1 is set and the current font is an image font, MS OS/2 ignores the current shear, angle, and box attributes. If mode 2 is set and the current font is an image font, MS OS/2 uses the current shear, angle, and box attributes. If mode 3 is set and the current font is an image font, MS OS/2 issues an error message. If the current font is a vector font, MS OS/2 always uses the current shear, angle, and box attributes (regardless of the mode).

sizfxCell Specifies the character-cell size (in world units). This **SIZEF** structure contains two fixed values.

ptlAngle Points to the **POINTL** structure that contains the coordinates of the endpoint of the character-angle vector. The baseline of vector characters is drawn parallel to the character-angle vector.

ptlShear Points to the **POINTL** structure that contains the coordinates of the endpoint of the character-shear vector. The vertical strokes in vector characters are drawn parallel to the character-shear vector.

usDirection Specifies the character direction. The default direction is from left to right. This parameter can be one of the following values:

Value	Meaning
CHDIRN_LEFTRIGHT	Left to right
CHDIRN_RIGHTLEFT	Right to left
CHDIRN_TOPBOTTOM	Top to bottom
CHDIRN_BOTTOMTOP	Bottom to top

See Also

GpiQueryAttrs, GpiQueryCharAngle, GpiQueryCharBox, GpiQueryCharSet, GpiQueryCp, GpiSetAttrs, GpiSetCharAngle, GpiSetCharBox, GpiSetCharSet, GpiSetCp, POINTL, SIZEF

Changes

The following character directions can now be specified for the **usDirection** field:

Value	Meaning
CHDIRN_LEFTRIGHT	Left to right
CHDIRN_RIGHTLEFT	Right to left
CHDIRN_TOPBOTTOM	Top to bottom
CHDIRN_BOTTOMTOP	Bottom to top

■ DENA1 New

```
typedef struct _DENA1 {     /* dena */
    UCHAR  reserved;
    UCHAR  cbName;
    USHORT cbValue;
    UCHAR  szName[1];
} DENA1;
```

The **DENA1** structure contains the names of the extended attributes returned by the **DosEnumAttribute** function.

Fields

reserved Specifies a reserved value; must be zero.

cbName Specifies the length of the extended-attribute name.

cbValue Specifies the length of the extended-attribute value.

szName[1] Contains the name of the extended attribute.

See Also

DosEnumAttribute

■ EAOP New

```
typedef struct _EAOP {      /* eaop */
    PGEALIST fpGEAList;
    PFEALIST fpFEAList;
    ULONG    oError;
} EAOP;
```

The **EAOP** structure contains extended-attribute information needed by the file-system function calls.

Fields

fpGEAList Points to the **GEALIST** structure that lists the extended attributes to retrieve.

fpFEAList Points to the **FEALIST** structure that lists the extended attributes found.

oError Specifies the offset, from the beginning of the structure, at which an error occurred.

See Also

DosFindFirst2, DosMkDir2, DosOpen2, DosQFileInfo, DosQPathInfo, DosSet-FileInfo, DosSetPathInfo, FEALIST, GEALIST

■ ENTRYFDATA New

```
typedef struct _ENTRYFDATA {      /* efd */
    USHORT cb;
    USHORT cchEditLimit;
    USHORT ichMinSel;
    USHORT ichMaxSel;
} ENTRYFDATA;
```

The **ENTRYFDATA** structure contains control data used to specify the characteristics of an entry-field control.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

cchEditLimit Specifies the maximum number of characters than can be entered in the edit control.

ichMinSel Specifies the beginning point of the current selection within the entry field's text buffer.

ichMaxSel Specifies the end point of the current selection within the entry field's text buffer.

■ FATTRS Change

```
typedef struct _FATTRS {    /* fat */
    USHORT  usRecordLength;
    USHORT  fsSelection;
    LONG    lMatch;
    CHAR    szFacename[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lMaxBaselineExt;
    LONG    lAveCharWidth;
    USHORT  fsType;
    USHORT  fsFontUse;
} FATTRS;
```

The **FATTRS** structure specifies the attributes of the logical font to be created by the **VioCreateLogFont** or **GpiCreateLogFont** function.

Fields

usRecordLength Specifies the length of the structure.

fsSelection Specifies one or more character attributes. This field can be any combination of the following values:

Value	Meaning
FATTR_SEL_ITALIC	Specifies italic characters.
FATTR_SEL_OUTLINE	Specifies an outline font.
FATTR_SEL_STRIKEOUT	Specifies strikeout characters.
FATTR_SEL_UNDERSCORE	Specifies underscored characters.
FATTR_SEL_BOLD	Specifies bold characters.

lMatch Specifies the match number for a specific font. The **VioQueryFonts** and **GpiQueryFonts** functions return a unique match number for each font. When this number is specified in the **lMatch** field, the specified font is used. If the **lMatch** field is zero, the system determines which font gives the best mapping to the required attributes.

szFacename[FACESIZE] Specifies the typeface name of the font.

idRegistry Specifies the registry number of the font.

usCodePage Specifies the code-page identifier of the font.

lMaxBaselineExt Specifies the sum of the maximum ascender and descender values for a font.

lAveCharWidth Specifies the average width of a character in a font. This value is obtained by multiplying the width of each lowercase letter by a weighted factor, adding the results for all of the letters in the alphabet, and dividing by 1000. The factor corresponds to the frequency of use for a particular letter. For example, the letter *e* appears frequently in text while the letter *q* does not; therefore, the factor assigned to *e* would be greater than the factor assigned to *q*.

fsType Specifies the type of the font. This field can include one or more of the following values:

Value	Meaning
FATTR_TYPE_KERNING	Specifies a kerned font.
FATTR_TYPE_MBCS	Specifies a multiple-byte character-set font.
FATTR_TYPE_DBCS	Specifies a double-byte character-set font.
FATTR_TYPE_ANTIALIASED	Specifies an anti-aliased font.

fsFontUse Specifies how the font is related to the character attributes. This field can be any combination of the following values:

Value	Meaning
FATTR_FONTUSE_NOMIX	The application cannot mix text and graphics.
FATTR_FONTUSE_OUTLINE	Requests an outline font.
FATTR_FONTUSE_TRANSFORMABLE	Requests a transformable font.

See Also

GpiCreateLogFont, GpiQueryFonts, VioCreateLogFont, VioQueryFonts

Changes

FATTR_TYPE_FIXED can no longer be specified for the **fsType** field. The following new constants can be specified for **fsType**:

Value	Meaning
FATTR_TYPE_MBCS	Specifies a multiple-byte character-set font.
FATTR_TYPE_DBCS	Specifies a double-byte character-set font.
FATTR_TYPE_ANTIALIASED	Specifies an anti-aliased font.

FATTR_SEL_OUTLINE can be specified for the **fsSelection** field.

Corrections

The FATTR_SEL_HOLLOW constant did not exist in the include files. A new constant, FATTR_SEL_OUTLINE, gives you hollow (outlined) characters.

■ FEA — New

```
typedef struct _FEA {      /* fea */
    BYTE   fEA;
    BYTE   cbName;
    USHORT cbValue;
} FEA;
```

The **FEA** structure contains the values of extended attributes.

Fields

fEA Specifies one or more flags. In MS OS/2, version 1.2, the only flag available is FEA_NEEDEA, indicating an extended-attribute bit is needed.

cbName Specifies the length of the extended-attribute name, not including the null terminating character.

cbValue Specifies the length of the extended-attribute value.

Comments

This structure also contains a variable-length portion immediately following the **cbValue** field. This variable-length portion contains the extended-attribute name and the extended-attribute value.

See Also

EAOP, FEALIST, GEA, GEALIST

■ FEALIST — New

```
typedef struct _FEALIST {     /* feal */
    ULONG cbList;
    FEA   list[1];
} FEALIST;
```

The **FEALIST** structure contains one or more extended attributes.

Fields

cbList	Specifies the size (in bytes) of the structure.
list[1]	Contains an array of one or more **FEA** structures.

Comments

The **FEALIST** structure contains a list of the extended attributes that were found. The **GEALIST** structure contains names of extended attributes to retrieve information for.

See Also

DosFindFirst2, DosMkDir2, DosOpen2, DosQPathInfo, DosSetFileInfo, DosSetPathInfo, EAOP, FEA, GEALIST

■ FILEFINDBUF2 — New

```
typedef struct _FILEFINDBUF2 {     /* findbuf2 */
    FDATE  fdateCreation;
    FTIME  ftimeCreation;
    FDATE  fdateLastAccess;
    FTIME  ftimeLastAccess;
    FDATE  fdateLastWrite;
    FTIME  ftimeLastWrite;
    ULONG  cbFile;
    ULONG  cbFileAlloc;
    USHORT attrFile;
    ULONG  cbList;
    UCHAR  cchName;
    CHAR   achName[CCHMAXPATHCOMP];
} FILEFINDBUF2;
```

The **FILEFINDBUF2** structure contains information about a file.

Fields

fdateCreation	Specifies the date the file was created.
ftimeCreation	Specifies the time the file was created.
fdateLastAccess	Specifies the date the file was last accessed.
ftimeLastAccess	Specifies the time the file was last accessed.
fdateLastWrite	Specifies the date the file was last written to.
ftimeLastWrite	Specifies the time the file was last written to.

cbFile Specifies the end of file data.

cbFileAlloc Specifies the allocated file size.

attrFile Specifies the file attributes.

cbList Specifies the size (in bytes) of the buffer needed for the list of extended attributes in a FIL_QUERYEASFROMLIST level request (see **DosFindFirst2**).

cchName Specifies the length of the null-terminated filename.

achName[CCHMAXPATHCOMP] Specifies the null-terminated filename.

See Also **DosFindFirst2, DosFindNext2, FDATE, FTIME**

■ FILESTATUS2 New

```
typedef struct _FILESTATUS2 {     /* fsts2 */
    FDATE   fdateCreation;
    FTIME   ftimeCreation;
    FDATE   fdateLastAccess;
    FTIME   ftimeLastAccess;
    FDATE   fdateLastWrite;
    FTIME   ftimeLastWrite;
    ULONG   cbFile;
    ULONG   cbFileAlloc;
    USHORT  attrFile;
    ULONG   cbList;
} FILESTATUS2;
```

The **FILESTATUS2** structure contains information about the status of a file.

Fields

fdateCreation Specifies the date the file was created.

ftimeCreation Specifies the time the file was created.

fdateLastAccess Specifies the date the file was last accessed.

ftimeLastAccess Specifies the time the file was last accessed.

fdateLastWrite Specifies the date the file was last written to.

ftimeLastWrite Specifies the time the file was last written to.

cbFile Specifies the end of file data.

cbFileAlloc Specifies the allocated file size.

attrFile Specifies the file attributes.

cbList Specifies the size of the extended-attribute buffer.

Comments The **cbFile**, **cbFileAlloc**, and **attrFile** fields are not used by the **DosSetFileInfo** function.

See Also **DosQFileInfo, DosQPathInfo, DosSetFileInfo**

FIOLOCKCMD New

```
typedef struct _FIOLOCKCMD {    /* flc */
    USHORT  usCmd;
    USHORT  cLockCnt;
    ULONG   cTimeOut;
} FIOLOCKCMD;
```

The **FIOLOCKCMD** structure contains information used by the **DosFileIO** function for locking a file.

Fields

usCmd Specifies the command to pass to the **DosFileIO** function. This field should be set to FIO_LOCK.

cLockCnt Specifies the number of **FIOLOCKREC** structures that follow this structure. An **FIOLOCKREC** structure specifies the area of the file to lock and whether another process can read the locked portion.

cTimeOut Specifies the time-out period (in milliseconds). If this field is NULL, the **DosFileIO** function continues immediately with the next command. If this field is - 1, **DosFileIO** waits indefinitely for the requested lock to become available. Any other value specifies the maximum amount of time **DosFileIO** waits for the requested lock to become available.

See Also **DosFileIO, DosFileLocks, FIOLOCKREC**

FIOLOCKREC New

```
typedef struct _FIOLOCKREC {    /* flr */
    USHORT  fShare;
    ULONG   cbStart;
    ULONG   cbLength;
} FIOLOCKREC;
```

The **FIOLOCKREC** structure contains information used by the **DosFileIO** function for locking a file. This structure is preceded by a **FIOLOCKCMD** structure that specifies the number of **FIOLOCKREC** structures to be used.

Fields

fShare Specifies whether other processes can read the portion of the file that is locked. A value of FIO_SHAREREAD allows other processes to read the file; a value of FIO_NOSHARE prevents other processes from reading the file.

cbStart Specifies the offset of the lock region. The offset is established from the beginning of the file.

cbLength Specifies the length (in bytes) of the region to be locked.

See Also **DosFileIO, FIOLOCKCMD**

■ FIOREADWRITE New

```
typedef struct _FIOREADWRITE {    /* frwc */
    USHORT usCmd;
    PVOID  pbBuffer;
    USHORT cbBufferLen;
    USHORT cbActualLen;
} FIOREADWRITE;
```

The **FIOREADWRITE** structure contains information used by the **DosFileIO** function for reading and writing data.

Fields

usCmd Specifies the command to pass to the **DosFileIO** function. This field should be set to FIO_READ for a read operation or to FIO_WRITE for a write operation.

pbBuffer Points to the buffer that contains the data to be written, or points to a buffer that receives the data that is read.

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

cbActualLen Specifies the number of bytes actually transferred.

See Also **DosFileIO**

■ FIOSEEKCMD New

```
typedef struct _FIOSEEKCMD {    /* fsc */
    USHORT  usCmd;
    USHORT  fsMethod;
    ULONG   cbDistance;
    ULONG   cbNewPosition;
} FIOSEEKCMD;
```

The **FIOSEEKCMD** structure contains information used by the **DosFileIO** function's seek operation.

Fields

usCmd Specifies the command to be passed to the **DosFileIO** function. This field must be set to FIO_SEEK.

fsMethod Specifies where to begin the seek operation. This field can be one of the following values:

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

cbDistance Specifies the new position requested for the file pointer. The value of this field is the number of bytes offset from the starting position specified in the **fsMethod** field.

cbNewPosition On return from the **DosFileIO** function, this field contains the new position of the file pointer relative to the beginning of the file.

See Also **DosChgFilePtr, DosFileIO**

■ FIOUNLOCKCMD New

```
typedef struct _FIOUNLOCKCMD {    /* fuc */
    USHORT  usCmd;
    USHORT  cUnlockCnt;
} FIOUNLOCKCMD;
```

The **FIOUNLOCKCMD** structure contains information used by the **DosFileIO** function for unlocking a file.

Fields

usCmd Specifies the command to pass to the **DosFileIO** function. This field must be set to FIO_UNLOCK.

cUnlockCnt Specifies the number of **FIOUNLOCKREC** structures that follow this structure.

See Also **DosFileIO, DosFileLocks, FIOUNLOCKREC**

■ FIOUNLOCKREC New

```
typedef struct _FIOUNLOCKREC {    /* fur */
    ULONG   cbStart;
    ULONG   cbLength;
} FIOUNLOCKREC;
```

The **FIOUNLOCKREC** structure contains information used by the **DosFileIO** function for unlocking a file. This structure is preceded by an **FIOUNLOCKCMD** structure that specifies the number of **FIOUNLOCKREC** structures that are used.

Fields

cbStart Specifies the offset of the unlock region. The offset is determined from the beginning of the file.

cbLength Specifies the length (in bytes) of the region to unlock.

See Also **DosFileIO, FIOUNLOCKCMD**

■ FONTMETRICS Change

```
typedef struct _FONTMETRICS {    /* fm */
    CHAR     szFamilyname[FACESIZE];
    CHAR     szFacename[FACESIZE];
    USHORT   idRegistry;
    USHORT   usCodePage;
    LONG     lEmHeight;
    LONG     lXHeight;
    LONG     lMaxAscender;
    LONG     lMaxDescender;
    LONG     lLowerCaseAscent;
    LONG     lLowerCaseDescent;
    LONG     lInternalLeading;
    LONG     lExternalLeading;
    LONG     lAveCharWidth;
    LONG     lMaxCharInc;
    LONG     lEmInc;
    LONG     lMaxBaselineExt;
    SHORT    sCharSlope;
    SHORT    sInlineDir;
    SHORT    sCharRot;
    USHORT   usWeightClass;
    USHORT   usWidthClass;
    SHORT    sXDeviceRes;
    SHORT    sYDeviceRes;
    SHORT    sFirstChar;
    SHORT    sLastChar;
    SHORT    sDefaultChar;
    SHORT    sBreakChar;
    SHORT    sNominalPointSize;
    SHORT    sMinimumPointSize;
    SHORT    sMaximumPointSize;
    USHORT   fsType;
    USHORT   fsDefn;
    USHORT   fsSelection;
    USHORT   fsCapabilities;
    LONG     lSubscriptXSize;
    LONG     lSubscriptYSize;
    LONG     lSubscriptXOffset;
    LONG     lSubscriptYOffset;
    LONG     lSuperscriptXSize;
    LONG     lSuperscriptYSize;
    LONG     lSuperscriptXOffset;
    LONG     lSuperscriptYOffset;
    LONG     lUnderscoreSize;
    LONG     lUnderscorePosition;
    LONG     lStrikeoutSize;
    LONG     lStrikeoutPosition;
    SHORT    sKerningPairs;
    SHORT    sFamilyClass;
    LONG     lMatch;
} FONTMETRICS;
```

The **FONTMETRICS** structure contains information about fonts.

Fields

szFamilyname[FACESIZE] Specifies the family name of the font. Examples of common family names in MS OS/2 version 1.1 are Courier, Helvetica, and Times.

szFacename[FACESIZE] Specifies the typeface name of the font. Examples of common typeface names are Courier, Helvetica, and Times.

idRegistry Specifies the registry number of the font. For MS OS/2 version 1.1, this value must be zero.

usCodePage Identifies the code page an application should use with a particular font. For MS OS/2 version 1.1, this value must be 850.

lEmHeight Specifies the average height of uppercase characters. The height is measured in world coordinates from the baseline to the top of the character.

lXHeight Specifies the average height of lowercase characters. The height is measured in world coordinates from the baseline to the top of the character.

lMaxAscender Specifies the maximum height of any character in the font. The height is measured in world coordinates from the baseline to the top of the character.

lMaxDescender Specifies the maximum depth of any character in the font. The depth is measured in world coordinates from the baseline to the bottom of the lowest character.

lLowerCaseAscent Specifies the maximum height of any lowercase character in the font. The height is measured in world coordinates from the baseline to the top of the ascender of the tallest lowercase character.

lLowerCaseDescent Specifies the maximum depth of any lowercase character in a font. The depth is measure in world coordinates from the baseline to the bottom of the descender on the lowest lowercase character.

lInternalLeading Specifies the amount of space reserved in the top of each character cell for accent marks. This metric is always given in world coordinates.

lExternalLeading Specifies the amount of space that should appear between adjacent rows of text. This metric is always given in world coordinates.

lAveCharWidth Specifies the average character width for characters in the font. The average character width is determined by multiplying the width of each lowercase character by a predetermined constant, adding the results, and then dividing by 1000. Letters and their predetermined constances are listed as follows:

a	64	j	3	s	56
b	14	k	6	t	71
c	27	l	35	u	31
d	35	m	20	v	10
e	100	n	56	w	18
f	20	o	56	x	3
g	14	p	17	y	18
h	42	q	4	z	2
i	63	r	49	space	166

lMaxCharInc Specifies the maximum increment between characters in the font.

lEmInc Specifies the width of an uppercase *M* in the font.

lMaxBaselineExt Specifies the sum of the maximum ascender and maximum descender values.

sCharSlope Specifies the angle (in degrees and minutes) between a vertical line and the upright strokes in characters in the font. The first nine bits of this value contain the degrees, the next six bits contain the minutes, and the last bit is reserved. The slope of characters in a normal font is zero; the slope of italic characters is nonzero.

sInlineDir Specifies an angle (in degrees and minutes, increasing clockwise) from the *x*-axis that the system uses when it draws a text string. The system draws each consecutive character from the text string in the inline direction. The inline-direction angle for a Swiss font is zero; the inline direction for a Hebrew font is 180.

sCharRot Specifies the angle (in degrees and minutes) between the baseline of characters in the font and the *x*-axis. This is the angle assigned by the font designer.

usWeightClass Specifies the thickness of the strokes that form the characters in the font. This field can be one of the following values:

Value	Meaning
1	Ultra-light
2	Extra-light
3	Light
4	Semi-light
5	Medium (normal)
6	Semi-bold
7	Bold
8	Extra-bold
9	Ultra-bold

usWidthClass Specifies the relative-aspect ratio of characters in the font in relation to the normal-aspect ratio for a font of this type. The following are the possible values:

Value	Description	Normal aspect ratio
1	Ultra-condensed	50%
2	Extra-condensed	2.5%
3	Condensed	75%
4	Semi-condensed	87.5%
5	Normal	100%
6	Semi-expanded	112.5%
7	Expanded	125%
8	Extra-expanded	50%
9	Ultra-expanded	200%

sXDeviceRes Specifies the horizontal resolution of the target device for which the font was originally designed. This value is given in pels per inch.

sYDeviceRes Specifies the vertical resolution of the target device for which the font was originally designed. This value is given in pels per inch.

sFirstChar Specifies the code point for the first character in the font.

sLastChar Specifies the code point for the last character in the font. This code point is an offset from the **sFirstChar** value.

sDefaultChar Specifies the code point for the default character in the font. This code point is an offset from the **sDefaultChar** value. The default character is the character the system uses when an application specifies a code point that is out of the range of a font's code page.

sBreakChar Specifies the code point for the space character in the font. This code point is an offset from the **sFirstChar** value.

sNominalPointSize Specifies the height of the font (in decipoints—each decipoint is 1/720 inch). The nominal point size is the point size the font was designed to be drawn.

sMinimumPointSize Specifies the mimimum height of the font (in decipoints). A font should not be reduced to a size smaller than the minimum point size.

sMaximumPointSize Specifies the maximum height of the font (in decipoints). A font should not be increased to a size larger than this value.

fsType Specifies the type of font. This field can be one or more of the following values:

Value	Meaning
FM_TYPE_FIXED	Font is fixed. Font is proportional if this value is not specified.
FM_TYPE_LICENSED	Font is licensed.
FM_TYPE_KERNING	Font has kerning information.
FM_TYPE_DBCS	Font is a double-byte character set.
FM_TYPE_MBCS	Font is a multiple-byte character set.
FM_TYPE_64K	Font requires more than 64K of memory.

fsDefn Specifies the definition of the font. This field can be one or more of the following values:

Value	Meaning
FM_DEFN_OUTLINE	Specifies an outline font (vector).
FM_DEFN_GENERIC	Specifies a generic font (raster or bitmapped).

fsSelection Specifies how the characters are to be drawn. This field can be one or more of the following values:

Value	Meaning
FM_SEL_ITALIC	Characters are italic.
FM_SEL_UNDERSCORE	Characters are underscored.
FM_SEL_NEGATIVE	Characters are drawn using negative images.
FM_SEL_OUTLINE	Characters are outlined.
FM_SEL_STRIKEOUT	Characters are overstruck.
FM_SEL_BOLD	Characters are bold.

fsCapabilities Specifies whether the characters in this font can be mixed with graphics. If this field is FM_CAP_NOMIX, the characters cannot be mixed with graphics; otherwise, they can be mixed with graphics.

lSubscriptXSize Specifies the horizontal side (in world coordinates) for subscripts in the font.

lSubscriptYSize Specifies the vertical size (in world coordinates) for subscripts in the font.

lSubscriptXOffset Specifies the horizontal offset from the left edge of the character cell.

lSubscriptYOffset Specifies the vertical offset from the character-cell baseline.

lSuperscriptXSize Specifies the horizontal size (in world coordinates) for superscripts in the font.

lSuperscriptYSize Specifies the vertical size (in world coordinates) for superscripts in the font.

lSuperscriptXOffset Specifies the horizontal offset from the left edge of the character cell.

lSuperscriptYOffset Specifies the vertical offset from the character-cell baseline.

lUnderscoreSize Specifies the width of the underscore (in world coordinates).

lUnderscorePosition Specifies the distance from the baseline to the underscore line (in world coordinates).

lStrikeoutSize Specifies the width of the overstrike (in world coordinates).

lStrikeoutPosition Specifies the position of the overstrike in relation to the baseline.

sKerningPairs Specifies the number of kerning pairs in the kerning-pair table for the font.

sFamilyClass Specifies the font-family class and subclass.

lMatch Specifies a long integer value that should be copied to the **FATTRS** structure when the **GpiCreateLogFont** function is called. (When this value is passed, the system must select a font that contains the metrics associated with this **lMatch** field.)

See Also **GpiCreateLogFont, GpiQueryFontMetrics, GpiQueryFonts, VioQueryFonts**

Changes New constants have been added for the **fsType**, **fsDefn**, and **fsSelection** fields.

The **sReserved** field has been replaced by the **sFamilyClass** field.

■ FSINFO Change

```
typedef struct _FSINFO {     /* fsinf */
    ULONG ulVSN;
    VOLUMELABEL vol;
} FSINFO;
```

The **FSINFO** structure contains information about the volume label of a disk.

Fields

ulVSN Specifies the serial number of the disk. If there is no serial number on the disk, this field is zero.

vol Specifies a **VOLUMELABEL** structure that will contain the name of the volume label.

See Also

DosQFSInfo, VOLUMELABEL

Changes

The fields **fdateCreation** and **ftimeCreation** worked only for MS OS/2, version 1.1. These fields have been replaced by the **ulVSN** field, which receives the serial number of the disk for MS OS/2, version 1.2.

■ FSQBUFFER New

```
typedef struct _FSQBUFFER {     /* fsqbf */
    USHORT  iType;
    USHORT  cbName;
    UCHAR   szName[1];
    USHORT  cbFSDName;
    UCHAR   szFSDName[1];
    USHORT  cbFSAData;
    UCHAR   rgFSAData[1];
} FSQBUFFER;
```

The **FSQBUFFER** structure contains information about the file system attached to a driver or device.

Fields

iType Specifies the type of device. This field can contain one of the following values:

Value	Type
FSAT_CHARDEV	Resident character device
FSAT_PSEUDODEV	Pseudo-character device
FSAT_LOCALDRV	Local drive
FSAT_REMOTEDRV	Remote drive attached to a file system

cbName Specifies the length of the drive or device name, not including the null terminating character.

szName[1] Specifies the drive or device name. The actual length of this field varies, depending on the length of the device name.

cbFSDName Specifies the length of the file-system name, not including the null terminating character.

szFSDName[1] Specifies the file-system name the drive or device is attached to. The actual length of this field varies depending on the length of the file-system name. This field contains only a null character if the device is a resident character device.

cbFSAData Specifies the length of the data returned by the file system.

rgFSAData[1] Specifies the data returned by the file system. The actual length and meaning of this field varies, depending on the file system that is attached.

Comments This structure should be used only as a guideline. Because it contains variable-length fields, it cannot be used directly to retrieve the data.

See Also **DosQFSAttach**

GEA New

```
typedef struct _GEA {    /* gea */
    BYTE cbName;
    CHAR szName[1];
} GEA;
```

The **GEA** structure contains an extended-attribute name.

Fields

cbName Specifies the length of the extended-attribute name contained in the **szName** field, not including the null terminating character.

szName[1] Contains the extended-attribute name.

See Also **EAOP, FEA, GEALIST**

GEALIST New

```
typedef struct _GEALIST {    /* geal */
    ULONG cbList;
    GEA   list[1];
} GEALIST;
```

The **GEALIST** structure contains one or more extended-attribute names.

Fields

cbList Specifies the size (in bytes) of the structure.

list[1] Contains an array of one or more **GEA** structures.

Comments The **GEALIST** structure contains a list of extended-attribute names to retrieve information for. The **FEALIST** structure contains a list of extended attributes that were found.

See Also **DosFindFirst2, DosMkDir2, DosOpen2, DosQPathInfo, DosSetFileInfo, DosSetPathInfo, EAOP, FEALIST, GEA**

■ HCINFO Correction

```
typedef struct _HCINFO {    /* hci */
    CHAR   szFormname[32];
    LONG   cx;
    LONG   cy;
    LONG   xLeftClip;
    LONG   yBottomClip;
    LONG   xRightClip;
    LONG   yTopClip;
    LONG   xPels;
    LONG   yPels;
    LONG   flAttributes;
} HCINFO;
```

The **HCINFO** structure contains information about the hard-copy capabilities of a device.

Fields

szFormname[32] Specifies the form name.

cx Specifies the form width (in millimeters).

cy Specifies the form height (top to bottom, in millimeters).

xLeftClip Specifies the left clip limit (in millimeters).

yBottomClip Specifies the bottom clip limit (in millimeters).

xRightClip Specifies the right clip limit (in millimeters).

yTopClip Specifies the top clip limit (in millimeters).

xPels Specifies the number of pels between the left and right clip limits.

yPels Specifies the number of pels between the top and bottom clip limits.

flAttributes Specifies whether the given form is the selected form. This field is HCAPS_CURRENT if the form is selected. Otherwise, it is zero.

See Also

DevQueryHardcopyCaps

Corrections

The **flAttributes** field is set to HCAPS_CURRENT when the specified form is the selected form.

■ HELPINIT New

```
typedef struct _HELPINIT   {  /* hinit */
  USHORT        cb;
  ULONG         ulReturnCode;
  PSZ           pszTutorialName;
  PHELPTABLE    phtHelpTable;
  HMODULE       hmodHelpTableModule;
  HMODULE       hmodAccelActionBarModule;
  USHORT        idAccelTable;
  USHORT        idActionBar;
  PSZ           pszHelpWindowTitle;
  USHORT        usShowPanelId;
  PSZ           pszHelpLibraryName;
} HELPINIT;
```

The **HELPINIT** structure is used when creating a help instance for an application.

Fields

cBytes Specifies the number of bytes in the initialization structure.

ulReturnCode Specifies the value returned by the system at initialization. A value of zero means that initialization was successful.

pszTutorialName Points to the string that contains the default tutorial name. If this field is NULL, the application does not have a tutorial or the tutorial name is specified in each help library.

phtHelpTable Points to the help table or to the resource ID of the help table. If you defined the table in a resource file, the low word should contain the resource ID of the table and the high word must be 0xFFFF.

hmodHelpTableModule Identifies the module handle returned by the **DosLoadModule** function when the application loaded the resource file. A value of NULL indicates that the resource file that contains the help table was appended to the application's executable (*.exe*) file.

hmodAccelActionBarModule Identifies the dynamic-link library that contains the accelerator table and menu-bar template used by a help window. A value of NULL indicates that the resource file containing the tailored accelerator table and menu bar was appended to the application's executable (*.exe*) file.

idAccelTable Identifies the accelerator table. The accelerator table is found in the dynamic-link library identified by the **hmodAccelActionBarModule** field. If the default accelerator table is to be used, this field should be NULL.

idActionBar Identifies the menu-bar template used by a help window. The menu-bar template is found in the dynamic-link library identified by the **hmodAccelActionBarModule** field. If the default menu bar is to be used, this field should be NULL.

pszHelpWindowTitle Points to the string that contains the window title of each help window.

usShowPanelId Specifies whether to display the window (panel) ID on a help window. If this value is CMIC_HIDE_PANEL_ID, the window ID is not shown; if this value is CMIC_SHOW_PANEL_ID, the window ID is shown.

pszHelpLibraryName Points to the string that contains the name of the help library that the system searches on each help request.

See Also

WinCreateHelpInstance, HELPTABLE

■ HELPTABLE New

```
typedef struct _HELPTABLE {      /* ht */
    USHORT          idAppWindow;
    PHELPSUBTABLE phstHelpSubTable;
    USHORT          idExtPanel;
} HELPTABLE;
```

The **HELPTABLE** structure identifies the help table for a specified application.

Fields

idAppWindow Specifies the window ID of a frame or dialog window.

phstHelpSubTable Points to a help subtable. The help subtable contains help panel IDs for the child windows and/or menus in the specified window.

idExtPanel Specifies an extended help panel ID. This help panel is displayed whenever extended help for the specified window is requested.

Comments

The help table for an application usually consists of an array of two or more HELPTABLE structures. Each structure specifies one window, such as a frame or dialog window, and points to one subtable containing the help panel IDs for each item in the window that the user may request help for. To mark the end of the array, the last structure in the array must be zero-filled.

The help subtable, pointed to by the **phstHelpSubTable** field, is an array help panel IDs and window or menu IDs. The first element in the help subtable, a 16-bit integer, specifies the size, in 16-bit words, of each subsequent element. The system requires that the first element be at least 2. All subsequent elements consist of the number of words specified by the first element. The first word in an element must be a window or menu ID. The second word must be a help panel ID. Any additional words are not used by the system. The last element in the help subtable must be zero-filled.

See Also

HM_CREATE_HELP_TABLE

■ KBDHWID New

```
typedef struct _KBDHWID {    /* kbhw */
    USHORT cb;
    USHORT idKbd;
    USHORT usReserved1;
    USHORT usReserved2;
} KBDHWID;
```

The **KBDHWID** structure contains information that identifies keyboard hardware.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

idKbd Specifies the ID number generated by the keyboard hardware. This field can be one of the following values:

Keyboard	Value
KEYBOARD_AT_COMPATABLE	IBM PC/AT or compatible keyboard
KEYBOARD_ENHANCED_101	101-key enhanced keyboard
KEYBOARD_ENHANCED_102	102-key enhanced keyboard
KEYBOARD_ENHANCED_122	122-key enhanced keyboard
KEYBOARD_SPACESAVER	Space Saver enhanced keyboard

usReserved1 Specifies a reserved value.

usReserved2 Specifies a reserved value.

See Also

KbdGetHWID

■ KBDKEYINFO Change

```
typedef struct _KBDKEYINFO {    /* kbci */
    UCHAR  chChar;
    UCHAR  chScan;
    UCHAR  fbStatus;
    UCHAR  bNlsShift;
    USHORT fsState;
    ULONG  time;
} KBDKEYINFO;
```

The **KBDKEYINFO** structure contains information about the last key pressed.

Fields

chChar Specifies the character derived from translation of the **chScan** field.

chScan Specifies the scan code received from the keyboard, identifying the key pressed. This scan code may be modified during the translation process.

fbStatus Specifies the state of the retrieved scan code. It can be any combination of the following values:

Value	Meaning
SHIFT_KEY_IN	Shift key is received (valid only in binary mode when shift reporting is turned on).
CONVERSION_REQUEST	Conversion requested.
FINAL_CHAR_IN	Final character received.
INTERIM_CHAR_IN	Interim character received.
EXTENDED_CODE	The scan code is an extended code, not a character.

bNlsShift Specifies a reserved value; must be zero.

fsState Specifies the state of the shift keys. It can be any combination of the following values:

Value	Meaning
RIGHTSHIFT	Right SHIFT key down.
LEFTSHIFT	Left SHIFT key down.
CONTROL	Either CONTROL key down.
ALT	Either ALT key down.
SCROLLLOCK_ON	SCROLL LOCK mode turned on.
NUMLOCK_ON	NUMLOCK mode turned on.
CAPSLOCK_ON	CAPSLOCK mode turned on.
INSERT_ON	INSERT key turned on.
LEFTCONTROL	Left CONTROL key down.
LEFTALT	Left ALT key down.
RIGHTCONTROL	Right CONTROL key down.
RIGHTALT	Right ALT key down.
SCROLLLOCK	SCROLL LOCK key down.

Value	Meaning
NUMLOCK	NUMLOCK key down.
CAPSLOCK	CAPSLOCK key down.
SYSREQ	SYSREQ key down.

time Specifies the time stamp of the keystroke (in milliseconds).

See Also **KbdCharIn**, **KbdPeek**, KBD_PEEKCHAR

Changes EXTENDED_CODE is a possible value for the **fsStatus** field and indicates the scan code is an extended code, not a character.

LDTADDRINFO New

```
typedef struct _LDTADDRINFO {  /* ldtaddr */
    PULONG  pulPhysAddr;
    USHORT  cb;
} LDTADDRINFO;
```

The **LDTADDRINFO** structure holds information about an address to be added to the local descriptor table (LDT).

Fields

pulPhysAddr Points to the 32-bit physical address of the beginning of the block of memory for which an LDT selector is requested.

cb Specifies the number of bytes for the requested memory.

See Also SCR_ALLOCLDT, SCR_ALLOCLDTOFF

LINFOSEG Change

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

The **LINFOSEG** structure contains information local to the current process.

Fields

pidCurrent Specifies the identifier of the current process.

pidParent Specifies the identifier of the parent process.

prtyCurrent Specifies the priority of the current thread.

tidCurrent Specifies the identifier of the current thread.

sgCurrent Specifies the current screen group.

rfProcStatus Specifies the process status. A value of PS_EXITLIST indicates the process is in an exit-list routine.

dummy1 Reserved.

fForeground Specifies that the current process is in foreground.

typeProcess Specifies the process type. It can be one of the following values:

Value	Meaning
PT_DETACHED	Process is running as a detached process.
PT_FULLSCREEN	Process is running in a full-screen protected-mode session.
PT_PM	Process is running in the Presentation Manager screen group.
PT_REALMODE	Process is running in DOS-compatibility mode.
PT_WINDOWABLEVIO	Process is running in a VIO-window session.

dummy2 Reserved.

selEnvironment Specifies the selector to the application's copy of the environment.

offCmdLine Specifies the offset to the environment where the command line that is used to run the current application is copied.

cbDataSegment Specifies the size of the default data segment.

cbStack Specifies the size of the stack.

cbHeap Specifies the size of the heap.

hmod Identifies the program.

selDS Specifies the default data segment.

Comments

The following fields are contained in registers at start up:

Field	Register
SelEnvironment	**ax**
offCmdLine	**bx**
cbDataSegment	**cx**
cbStack	**dx**
cbHeap	**si**
hmod	**di**
selDS	**ds**

See Also

DosGetInfoSeg, GINFOSEG

Changes

The PT_FULLSCREEN, PT_REALMODE, PT_WINDOWABLEVIO, PT_PM, and PT_DETACHED constants replace the numeric values previously defined for the **typeProcess** field. The constant PS_EXITLIST is a valid value for the **rfProcStatus** field.

Corrections

The **rfProcStatus** specifies the process status, not the subscreen group.

■ MATRIXLF — Correction

```
typedef struct _MATRIXLF {    /* matlf */
    FIXED fxM11;
    FIXED fxM12;
    LONG  lM13;
    FIXED fxM21;
    FIXED fxM22;
    LONG  lM23;
    LONG  lM31;
    LONG  lM32;
    LONG  lM33;
} MATRIXLF;
```

The **MATRIXLF** structure contains the scaling, translation, rotation, shear, and reflection transformation values that MS OS/2 uses when your application calls one of the transformation functions.

If the matrix contains scaling transformation values, the following fields are set:

Field	Description
fxM11	Specifies the horizontal scaling value.
fxM22	Specifies the vertical scaling value.

If the matrix contains translation transformation values, the following fields are set:

Field	Description
lM31	Specifies the horizontal translation value.
lM32	Specifies the vertical translation value.

If the matrix contains rotation transformation values, the following fields are set:

Field	Description
fxM11	Specifies the cosine of the angle of rotation.
fxM12	Specifies the negative sine of the angle of rotation.
fxM21	Specifies the sine of the angle of rotation.
fxM22	Specifies the cosine of the angle of rotation.

If the matrix contains vertical-shear transformation values, the following fields are set:

Field	Description
fxM21	Specifies the horizontal shear value.
fxM22	Specifies the vertical shear value.

If the matrix contains horizontal-shear transformation values, the following fields are set:

Field	Description
fxM11	Specifies the horizontal-shear value.
fxM12	Specifies the vertical-shear value.

If the matrix contains reflection values, the following fields are set:

Field	Description
fxM11	Specifies the vertical-reflection value. (This value is always negative. It causes reflection about the *x*-axis.)
fxM22	Specifies the horizontal-reflection value. (This value is always negative. It causes reflection about the *y*-axis.)

See Also

GpiCallSegmentMatrix, GpiQueryDefaultViewMatrix, GpiQueryModel-TransformMatrix, GpiQuerySegmentTransformMatrix, GpiQueryViewing-TransformMatrix, GpiSetDefaultViewMatrix, GpiSetModelTransformMatrix, GpiSetSegmentTransformMatrix, GpiSetViewingTransformMatrix

Corrections

If the matrix contains scaling transformation values, the **fxM22** field contains the vertical scaling value, not the **fxM12** field.

■ MLE_SEARCHDATA New

```
typedef struct _MLE_SEARCHDATA {    /* mlesrch */
    USHORT cb;
    PCHAR  pchFind;
    PCHAR  pchReplace;
    SHORT  cchFind;
    SHORT  cchReplace;
    IPT    iptStart;
    IPT    iptStop;
    USHORT cchFound;
} MLE_SEARCHDATA;
```

The **MLE_SEARCHDATA** structure contains information required to perform a search of a multiple-line entry field (MLE) using the MLM_SEARCH message.

Fields

cb Specifies the size of the structure (in bytes). The size depends on the operating-system version. Programs written in the C language should use the **sizeof** operator to set this field.

pchFind Points to the null-terminated string to find.

pchReplace Points to the null-terminated replacement string.

cchFind Specifies the number of characters to delete in the search string before inserting the replacement string. This field is used only if the MLFSEARCH_CHANGEALL flag is specified in the MLM_SEARCH message.

cchReplace Specifies the number of replacement-string characters to insert in the MLE text. This field is used only if the MLFSEARCH_CHANGEALL flag is specified in the MLM_SEARCH message.

iptStart Specifies the offset (number of characters from the beginning of the text) of the first character to search. A value of - 1 causes the search to start at the current cursor position.

iptStop Specifies the offset of the last character to search. A negative value causes the search to end at the end of the text.

cchFound Specifies the length (in characters) of the string found.

Comments

If the **iptStop** field is less than the **iptStart** field, the search wraps to the beginning of the text. If the two fields are identical, all the text in the MLE is searched.

See Also

MLM_SEARCH

MLECTLDATA New

```
typedef struct _MLECTLDATA {     /* mlectl */
    USHORT cbCtlData;
    USHORT afIEFormat;
    ULONG  cchText;
    IPT    iptAnchor;
    IPT    iptCursor;
    LONG   cxFormat;
    LONG   cyFormat;
    ULONG  afFormatFlags;
} MLECTLDATA;
```

The **MLECTLDATA** structure contains multiple-line entry-field (MLE) format information.

Fields

cbCtlData Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

afIEFormat Specifies the import/export format. This parameter is be one of the following values:

Value	Meaning
MLFIE_CFTEXT	Specifies the clipboard text format. This format uses carriage-return/linefeed characters for line breaks on export, and recognizes linefeed, carriage-return/linefeed, or linefeed/carriage-return characters for line breaks on import. This is the default format.
MLFIE_NOTRANS	Specifies a format that uses linefeed characters for line breaks. Guarantees that any text imported into the MLE in this form can be recovered in exactly the same form on export.

Value	Meaning
MLFIE_WINFMT	Specifies the format of the MLE window. This format recognizes carriage-return/linefeed characters for line breaks on import. It ignores the sequence carriage-return/carriage-return/linefeed. On export, it uses carriage-return/linefeed characters to denote a hard line break and carriage-return/carriage-return/linefeed character to denote a soft line break caused by word-wrapping.

cchText Specifies the maximum amount (in bytes) of text.

iptAnchor Specifies the offset (number of characters from the beginning of the text) of the first character of the selection.

iptCursor Specifies the offset of the cursor position (one character to the right of the selection).

cxFormat Specifies the width (in pels) of the format rectangle.

cyFormat Specifies the height (in pels) of the format rectangle.

afFormatFlags Specifies how the format rectangle is to be treated. This parameter can be one or more of the following flags:

Value	Meaning
MLFFMTRECT_LIMITHORZ	Specifies that the text within the MLE cannot exceed the horizontal dimension specified by the **cxFormat** field. If word-wrap mode is turned on when the format rectangle is set, lines automatically wrap to stay within the horizontal limit of the format rectangle. If word-wrap mode is turned off when the format rectangle is set, an MLN_PIXHORZOVERFLOW notification message is sent to the application whenever an operation would exceed the horizontal limit specified in the format rectangle.
MLFFMTRECT_LIMITVERT	Specifies that the text within the MLE cannot exceed the vertical dimension specified by the **cxFormat** field. Whenever an MLE operation would cause text to exceed the vertical limit, an MLN_PIXVERTOVERFLOW notification message is sent to the application.

Value	Meaning
MLFFMTRECT_MATCHWINDOW	Specifies that the format rectangle is to be kept the same size as the MLE window (minus the border or scroll bars).
MLFFMTRECT_FORMATRECT	Specifies that the format rectangle is to be kept the same size as the MLE window (minus the border or scroll bars) and that text cannot exceed the size of the window. This value is equivalent to combining the values MLFFMTRECT_LIMITHORZ, MLFFMTRECT_LIMITVERT, and MLFFMTRECT_MATCHWINDOW.

See Also MLM_FORMAT, MLM_SETFORMATRECT

■ MLEFORMATRECT New

```
typedef struct _MLEFORMATRECT {     /* mlefrd */
    LONG cxFormat;
    LONG cyFormat;
} MLEFORMATRECT;
```

The **MLEFORMATRECT** structure contains width and height information for the multiple-line entry-field (MLE) format rectangle.

Fields

cxFormat Specifies the width (in pels) of the MLE format rectangle. If this field is - 1, the current MLE-window width (minus any border or scroll bars) is used. If this field is 0, there is no limit on the MLE width.

cyFormat Specifies the height (in pels) of the format rectangle. If this field is - 1, the current MLE-window height (minus any border or scroll bars) is used. If this field is 0, there is no limit on the MLE height.

See Also MLM_QUERYFORMATRECT, MLM_SETFORMATRECT

■ MLEMARGSTRUCT New

```
typedef struct _MLEMARGSTRUCT {     /* mlemrg */
    USHORT afMargins;
    USHORT usMouMsg;
    IPT    iptNear;
} MLEMARGSTRUCT;
```

The **MLEMARGSTRUCT** structure contains data used by the MLN_MARGIN message to notify an application when the user moves the mouse to one of the margins.

Fields

afMargins Specifies the margin. This field can be one of the following values:

Value	Meaning
MLFMARGIN_LEFT	The mouse was moved over the left margin.
MLFMARGIN_RIGHT	The mouse was moved over the right margin.
MLFMARGIN_TOP	The mouse was moved over the top margin.
MLFMARGIN_BOTTOM	The mouse was moved over the bottom margin.

usMouMsg Specifies the mouse message associated with the move.

iptNear Specifies the offset (number of characters from the beginning of the text) of the character nearest to the mouse.

See Also MLN_MARGIN, WM_CONTROL

■ MLEOVERFLOW New

```
typedef struct _MLEOVERFLOW {    /* mleovr */
   ULONG afErrInd;
   LONG nBytesOver;
   LONG pixHorzOver;
   LONG pixVertOver;
}  MLEOVERFLOW;
```

The **MLEOVERFLOW** structure contains information about overflow in a multiple-line entry field (MLE).

Fields

afErrInd Specifies the cause of the error. This parameter can be one of the following values:

Value	Meaning
MLFEFR_RESIZE	The overflow was the result of a resize operation that overflowed a format rectangle.
MLFEFR_TABSTOP	The overflow was the result of resetting tab stops that overflowed a format rectangle.
MLFEFR_FONT	The overflow was the result of changing font information.
MLFEFR_TEXT	The overflow was the result of a text insertion operation with the format rectangle set.
MLFEFR_WORDWRAP	The overflow was the result of setting word wrap while the MLE text exceeds the format rectangle.
MLFETL_TEXTBYTES	The overflow was the result of a text insertion operation with the text limit set.

nBytesOver Specifies the number of bytes that overflowed.

pixHorzOver Specifies the number of pels that overflowed horizontally.

pixVertOver Specifies the number of pels that overflowed vertically.

See Also MLN_OVERFLOW, WM_CONTROL

■ PARAM New

```
typedef struct _PARAM {     /* param */
    ULONG id;
    ULONG cb;
    BYTE  ab[1];
} PARAM;
```

The **PARAM** structure contains a presentation parameter.

Fields

id Identifies the presentation parameter. It can be one of the following values:

Value	Meaning
PP_FOREGROUNDCOLOR	RGB foreground color
PP_FOREGROUNDCOLORINDEX	Color index of foreground color
PP_BACKGROUNDCOLOR	RGB background color
PP_BACKGROUNDCOLORINDEX	Color index of background color
PP_HILITEFOREGROUNDCOLOR	RGB color of foreground highlighted area
PP_HILITEFOREGROUNDCOLORINDEX	Color index of foreground highlighted area
PP_HILITEBACKGROUNDCOLOR	RGB color of background highlighted area
PP_HILITEBACKGROUNDCOLORINDEX	Color index of background highlighted area
PP_DISABLEDFOREGROUNDCOLOR	RGB foreground disabled color
PP_DISABLEDFOREGROUNDCOLORINDEX	Color index of foreground disabled color
PP_DISABLEDBACKGROUNDCOLOR	RGB color of background disabled color
PP_DISABLEDBACKGROUNDCOLORINDEX	Color index of background disabled color
PP_BORDERCOLOR	RGB color of window border
PP_BORDERCOLORINDEX	Color index of window border

Value	Meaning
PP_FONTNAMESIZE	Font size
PP_FONTHANDLE	Font handle

A value of zero for this parameter specifies an application-defined string.

cb Specifies the length of the presentation parameter.

ab[1] Specifies an array of bytes containing the presentation parameter.

See Also **PRESPARAMS**

■ PIPESEMSTATE New

```
typedef struct _PIPESEMSTATE {    /* nmpsmst */
    BYTE   fStatus;
    BYTE   fFlag;
    USHORT usKey;
    USHORT usAvail;
} PIPESEMSTATE;
```

The **PIPESEMSTATE** structure contains named-pipe information retrieved by using the **DosQNmPipeSemState** function.

Fields

fStatus Specifies the status of the named pipe. This field can be one of the following values:

Value	Meaning
NPSS_EOI	End of information.
NPSS_RDATA	Readable data is available.
NPSS_WSPACE	Write space is available.
NPSS_CLOSE	Pipe is in closing state.

fFlag Specifies additional information. If this field is NPSS_WAIT, there is a waiting thread on the end of the pipe.

usKey Specifies the user's key value.

usAvail Specifies the available data if the **fStatus** field is NPSS_RDATA, or the available space if the **fStatus** field is NPSS_WSPACE.

See Also **DosQNmPipeSemState**

■ PRESPARAMS New

```
typedef struct _PRESPARAMS {    /* pres */
    ULONG cb;
    PARAM aparam[1];
} PRESPARAMS;
```

The **PRESPARAMS** structure contains an array of **PARAM** structures that contain presentation parameters.

Fields

cb Specifies the size (in bytes) of the structure, including the array of **PARAM** structures.

aparam[1] Specifies an array of one or more **PARAM** structures.

See Also

PARAM

■ PRFPROFILE New

```
typedef struct _PRFPROFILE {     /* prfpro */
    ULONG  cchUserName;
    PSZ    pszUserName;
    ULONG  cchSysName;
    PSZ    pszSysName;
} PRFPROFILE;
```

The **PRFPROFILE** structure specifies the names of files that contain profile information.

Fields

cchUserName Specifies the number of characters in the string pointed to by the **pszUserName** field.

pszUserName Points to the null-terminated string that contains the name of the file used to store user-profile information.

cchSysName Specifies the number of characters in the string pointed to by the **pszSysName** field.

pszSysName Points to the null-terminated string that contains the name of the file used to store system-profile information.

See Also

PrfQueryProfile, PrfReset

■ PROGDETAILS New

```
typedef struct _PROGDETAILS {     /* progde */
    ULONG     Length;
    PROGTYPE  progt;
    USHORT    pad1[3];
    PSZ       pszTitle;
    PSZ       pszExecutable;
    PSZ       pszParameters;
    PSZ       pszStartupDir;
    PSZ       pszIcon;
    PSZ       pszEnvironment;
    SWP       swpInitial;
    USHORT    pad2[5];
} PROGDETAILS;
```

The **PROGDETAILS** structure contains information about a program.

Fields

Length Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

progt Specifies the **PROGTYPE** structure that contains program-type information.

pad1[3] Reserved.

pszTitle Points to the null-terminated string that contains the program title. This string must not exceed MAXNAMEL (defined in the include files) characters plus the terminating NULL character.

pszExecutable Points to the null-terminated string that contains the name of the executable file. If the string appears to be a fully qualified path (that is, it contains a colon in the second position and/or contains a backslash), the file is searched for in the indicated directory on the indicated drive. If neither of these conditions is true and the file is not in the current directory, each drive and directory specified in the path defined in the current program's environment is searched.

pszParameters Points to the null-terminated string that contains any parameters to pass to the program.

pszStartupDir Points to the null-terminated string that contains the default drive and directory.

pszIcon Points to the null-terminated string that contains the name of an icon file. This parameter is not used for MS OS/2, version 1.2.

pszEnvironment Points to the string that contains the environment variables. Each string is null-terminated, with the final string ending with two NULL characters.

swpInitial Specifies the **SWP** structure that contains the initial state of the program's window. If the **cy**, **cx**, **y**, and **x** fields of this structure are zero, a default window size is used when the application is started.

pad2[5] Reserved.

See Also **PrfAddProgram, PrfChangeProgram, PrfQueryDefinition, PROGTYPE, SWP**

■ PROGTITLE New

```
typedef struct _PROGTITLE {    /* progti */
    HPROGRAM hprog;
    PROGTYPE progt;
    USHORT   pad1[3];
    PSZ      pszTitle;
} PROGTITLE;
```

The **PROGTITLE** structure is used to specify program-title information.

Fields

hprog Specifies the handle of the program.

progt Specifies the **PROGTYPE** structure that contains program-type information.

pad1[3] Reserved.

pszTitle Points to the string that contains the program title.

See Also **PrfQueryProgramTitles, PROGTYPE**

PROGTYPE — Change

```
typedef struct _PROGTYPE {      /* progt */
    PROGCATEGORY progc;
    BYTE         fbVisible;
} PROGTYPE;
```

The **PROGTYPE** structure is used in the **PIBSTRUCT** and **PROGDETAILS** structures to specify a program or group type.

Fields

progc Specifies the program category. This field can be one of the following values:

Value	Meaning
PROG_DEFAULT	Default category.
PROG_FULLSCREEN	Program runs only in a full-screen session.
PROG_WINDOWABLEVIO	Program runs in a VIO-window session.
PROG_PM	Program is a Presentation Manager application.
PROG_GROUP	Handle is to a group.
PROG_REAL	Program is a (DOS) real-mode application.
PROG_DLL	Program is a dynamic-link library.

fbVisible Specifies the visibility of a program and (optionally) the protected or unprotected state of a group. This flag can be a combination of the following values:

Value	Meaning
SHE_VISIBLE	The program or group is visible.
SHE_INVISIBLE	The program or group is invisible and cannot be viewed.
SHE_UNPROTECTED	The group is unprotected. Programs can be added or deleted from the group. This value is valid only for groups.
SHE_PROTECTED	The group is protected. Programs cannot be added or deleted from the group; the only program information that can be changed is the visibility state. This value is valid only for groups.

See Also

PIBSTRUCT, PROGDETAILS

Changes

The **fbVisible** field has two additional options (SHE_UNPROTECTED and SHE_PROTECTED) that can be set when the structure is used to create or change a group. The program category PROG_DLL has also been added.

■ PTRACEBUF Change

```
typedef struct _PTRACEBUF {     /* ptrcbf */
    PID    pid;
    TID    tid;
    USHORT cmd;
    USHORT value;
    USHORT offv;
    USHORT segv;
    USHORT mte;
    USHORT rAX;
    USHORT rBX;
    USHORT rCX;
    USHORT rDX;
    USHORT rSI;
    USHORT rDI;
    USHORT rBP;
    USHORT rDS;
    USHORT rES;
    USHORT rIP;
    USHORT rCS;
    USHORT rF;
    USHORT rSP;
    USHORT rSS;
} PTRACEBUF;
```

The **PTRACEBUF** structure contains various debugging information.

Fields

pid Specifies the process identifier of the program being debugged.

tid Specifies the thread identifier of the program being debugged.

cmd Specifies the command to carry out. It can be one of the following values:

Value	Meaning
0x0001	Read memory instruction space (I-space).
0x0002	Read memory data space (D-space).
0x0003	Read registers.
0x0004	Write memory I-space.
0x0005	Write memory D-space.
0x0006	Write registers.
0x0007	Begin execution.
0x0008	Terminate child process.
0x0009	Single step.
0x000A	Suspend child process.
0x000B	Freeze child process.
0x000C	Resume child process.
0x000D	Convert segment number to selector.
0x000E	Get floating-point registers. The **segv** and **offv** fields must specify the address of a 94-byte buffer that receives the floating-point register values.
0x000F	Set floating-point registers. The **segv** and **offv** fields must specify the address of a 94-byte buffer that contains the floating-point register values.

Value	Meaning
0x0010	Get library-module name. The **value** field must contain the handle of the library module. The **segv** and **offv** fields must contain the address of the buffer that receives the name. This command should be used instead of the **DosGetModHandle** and **DosGetModName** functions to verify the name of a library loaded by the program being debugged.
0x0011	Get the thread identifier of the next thread. This field is circular; to read the registers of all threads in the process, use this value until a thread identifier is repeated. For more information about this value, see the "Comments" section.

When the command identified in the **cmd** field returns, it copies a code to the **value** field that specifies 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. The fault type is in the **value** field. The **segv** and **offv** fields contain the address that caused the fault.
0xFFF8	Library module has just been loaded. The **value** field contains the library-module handle.
0xFFF7	Process has not used 287 yet.
0xFFF6	Thread ending.
0xFFF5	Asynchronous stop.

value Specifies the value to be used for a given command or a return value from a command. If an error occurs, this 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 the segment selector.

mte Identifies the handle of the module that contains the segment.

rAX Specifies the **ax** register.

rBX Specifies the **bx** register.

rCX Specifies the **cx** register.

rDX Specifies the **dx** register.

rSI Specifies the **si** register.

rDI Specifies the **di** register.

rBP Specifies the **bp** register.

rDS Specifies the **ds** register.

rES Specifies the **es** register.

rIP Specifies the **ip** register.

rCS Specifies the **cs** register.

rF Specifies flags.

rSP Specifies the **sp** register.

rSS Specifies the **ss** register.

Comments

The 0x0011 value in the **cmd** field causes a thread identifier to be retrieved. The status of this thread is returned in a **ThreadStatus** buffer pointed to by the **segv** and **offv** fields. The format of the **ThreadStatus** buffer is as follows:

```
struct ThreadStatus {
    UCHAR  fDebugState;
    UCHAR  fThreadState;
    USHORT usThreadPriority;
    };
```

The **DebugState** field contains one of the following values:

Value	Meaning
0x0	Thread not frozen by debugger.
0x1	Thread frozen by debugger.

The **ThreadState** field contains one of the following:

Value	Meaning
0x0	Thread can be run.
0x1	Thread is suspended.
0x2	Thread is blocked.
0x3	Thread state is a critical section.

The **ThreadPriority** field receives the priority of the specified thread. The high byte receives the priority class, and the low byte receives the priority level.

See Also

DosGetModHandle, DosGetModName, DosPTrace

Changes

An additional value, 0x0011, can be specified for the **cmd** field. Two additional values, 0xFFF6 and 0xFFF5, can be returned in the **cmd** field.

■ PTRDRAWDATA New

```
typedef struct _PTRDRAWDATA { /* ptrdd */
    USHORT cb;
    USHORT usConfig;
    USHORT usFlag;
} PTRDRAWDATA;
```

The **PTRDRAWDATA** structure contains data for drawing the pointer.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

usConfig Specifies the display configuration. It can be one of the following values:

Value	Meaning
VIO_CONFIG_CURRENT	The current display adapter
VIO_CONFIG_PRIMARY	The primary display adapter
VIO_CONFIG_SECONDARY	The secondary display adapter

usFlag Specifies a flag that determines if this configuration is for an application or the base video subsystem (BVS). A value of 0x0000 specifies an application; 0x0001 specifies the BVS.

See Also

MOU_SETPROTDRAWADDRESS

■ SBCDATA Change

```
typedef struct _SBCDATA {     /* sbcd */
    USHORT cb;
    USHORT sHilite;
    SHORT  posFirst;
    SHORT  posLast;
    SHORT  posThumb;
    SHORT  cVisible;
    SHORT  cTotal;
} SBCDATA;
```

The **SBCDDATA** structure contains information about a scroll-bar window.

Fields

cb Specifies the size of the structure (in bytes). The size depends on the version of the operating system. Programs written in the C language should use the **sizeof** operator to set this field.

sHilite reserved, should be set to zero

posFirst Specifies the first possible position of the slider bar.

posLast Specifies the last possible position of the slider bar.

posThumb Specifies the current position of the slider bar.

cVisible Specifies the number of items (lines in a file, rows on a spreadsheet, etc) that are visible in the window.

cTotal Specifies the total number of items to be displayed.

Changes

The fields **cVisible** and **cTotal** have been added.

■ STATUSDATA — Correction

```
typedef struct _STATUSDATA {     /* stsdata */
    USHORT Length;
    USHORT SelectInd;
    USHORT BondInd;
} STATUSDATA;
```

The **STATUSDATA** structure contains status information about a session.

Fields

Length Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

SelectInd Specifies whether the target session should be set as selectable or nonselectable. It can be one of the following values:

Value	Meaning
TARGET_UNCHANGED	Leave current setting unchanged.
TARGET_SELECTABLE	Set as selectable.
TARGET_NOT_SELECTABLE	Set as nonselectable.

BondInd Specifies which session to bring to the foreground the next time the parent session is selected. It can be one of the following values:

Value	Meaning
BOND_UNCHANGED	Leave current setting unchanged.
BOND_CHILD	A bond between the parent session and the child session is established. The child session is brought to the foreground the next time the parent session is selected. If the child session is selected, the child session is brought to the foreground.
BOND_NONE	Any bond previously established with the specified child session is broken. The parent session is brought to the foreground the next time the parent session is selected and the child session is brought to the foreground the next time the child session is selected.

See Also **DosSetSession**

Corrections The third field is **BondInd**, not **BindInd**. Accordingly, the three constants have been changed to BOND_.

■ SWBLOCK — New

```
typedef struct _SWBLOCK {     /* swblk */
    USHORT  cswentry;
    SWENTRY aswentry[1];
} SWBLOCK;
```

The **SWBLOCK** structure contains an array of **SWENTRY** structures that contain information about the programs in the Task List.

Fields

cswentry Specifies the number of **SWENTRY** structures contained in the **aswentry** field.

aswentry[1] Contains an array of **SWENTRY** structures.

See Also

WinQuerySwitchList, SWENTRY

■ TRACKINFO — Change

```
typedef struct _TRACKINFO {    /* ti */
    SHORT  cxBorder;
    SHORT  cyBorder;
    SHORT  cxGrid;
    SHORT  cyGrid;
    SHORT  cxKeyboard;
    SHORT  cyKeyboard;
    RECTL  rclTrack;
    RECTL  rclBoundary;
    POINTL ptlMinTrackSize;
    POINTL ptlMaxTrackSize;
    USHORT fs;
    USHORT cxLeft;
    USHORT cyBottom;
    USHORT cxRight;
    USHORT cyTop;
} TRACKINFO;
```

The **TRACKINFO** structure contains information about a tracking rectangle used by the **WinTrackRect** function.

Fields

cxBorder Specifies the border width.

cyBorder Specifies the border height.

cxGrid Specifies the horizontal bounds of the tracking movements.

cyGrid Specifies the vertical bounds of the tracking movements.

cxKeyboard Specifies the amount of horizontal movement that occurs when the user presses the left arrow key.

cyKeyboard Specifies the amount of vertical movement that occurs when the user presses the left arrow key.

rclTrack Specifies the starting tracking rectangle. This is modified as the rectangle is tracked and holds the new tracking position when tracking is complete.

rclBoundary Specifies an absolute boundary for the tracking rectangle.

ptlMinTrackSize Specifies the minimum tracking size.

ptlMaxTrackSize Specifies the maximum tracking size.

fs Specifies tracking options. This field can be a combination of the following values:

Option	Meaning
TF_LEFT	Track the left side of the rectangle.
TF_TOP	Track the top side of the rectangle.
TF_RIGHT	Track the right side of the rectangle.
TF_BOTTOM	Track the bottom side of the rectangle.

Option	Meaning
TF_MOVE	Track all sides of the rectangle.
TF_SETPOINTERPOS	Repositions the pointer according to the other options specified.
TF_LEFT	Vertically centers the pointer at the left of the tracking rectangle.
TF_TOP	Horizontally centers the pointer at the top of the tracking rectangle.
TF_RIGHT	Vertically centers the pointer at the right of the tracking rectangle.
TF_BOTTOM	Horizontally centers the pointer at the bottom of the tracking rectangle.
TF_MOVE	Centers the pointer in the tracking rectangle.
TF_GRID	Restricts tracking to the grid defined by **cxGrid** and **cyGrid**.
TF_STANDARD	The width, height, grid-width and grid-height are all multiples of border-width and border-height.
TF_ALLINBOUNDARY	Performs tracking so that no part of the tracking rectangle ever falls outside the bounding rectangle.
TF_PARTINBOUNDARY	Performs tracking so that values of **cxLeft**, **cyBottom**, **cxRight**, and **cyTop** specify how much of the corresponding edge of the tracking rectangle must be kept within the opposite edge of the boundary rectangle.

cxLeft Specifies how much of the tracking rectangle must be kept within the boundary rectangle. Used only if **fs** is TF_PARTINBOUNDARY.

cyBottom Specifies how much of the tracking rectangle must be kept within the boundary rectangle. Used only if **fs** is TF_PARTINBOUNDARY.

cxRight Specifies how much of the tracking rectangle must be kept within the boundary rectangle. Used only if **fs** is TF_PARTINBOUNDARY.

cyTop Specifies how much of the tracking rectangle must be kept within the boundary rectangle. Used only if **fs** is TF_PARTINBOUNDARY.

See Also

WinTrackRect

Changes

The TF_PARTINBOUNDARY option can be used in the **fs** field.

Corrections

The TF_SETPOINTERPOS flag was incorrectly spelled TF_POINTERPOS.

The TF_ALLINBOUNDARY flag was incorrectly spelled TF_ALINBOUNDARY.

■ VIOCOLORREG New

```
typedef struct _VIOCOLORREG {    /* viocreg */
    USHORT  cb;
    USHORT  type;
    USHORT  firstcolorreg;
    USHORT  numcolorregs;
    PCH     colorregaddr;
} VIOCOLORREG;
```

The **VIOCOLORREG** structure contains the addresses of color registers.

Fields

cb Specifies the length of the structure (in bytes). The length determines how many color registers are retrieved.

type Specifies the request type. To retrieve the color registers, this field must be set to 0x0003.

firstcolorreg Specifies the first color register to be retrieved. This field must be a value from 0x0000 through 0x000F. The color registers are in sequential order. The number of registers retrieved depends on the structure size, as specified by the **cb** field.

numcolorregs Specifies the number of color registers to retrieve.

colorregaddr Points to the array that receives the color values for the registers. For each color-register retrieved, there should be three bytes allocated (one each for the red, green, and blue color values).

See Also

VioGetState, VioSetState

■ VIOCONFIGINFO Change

```
typedef struct _VIOCONFIGINFO {    /* vioin */
    USHORT  cb      ;
    USHORT  adapter;
    USHORT  display;
    ULONG   cbMemory;
    USHORT  Configuration;
    USHORT  VDHVersion;
    USHORT  Flags;
    ULONG   HWBufferSize;
    ULONG   FullSaveSize;
    ULONG   PartSaveSize;
    USHORT  EMAdaptersOFF;
    USHORT  EMDisplaysOFF;
} VIOCONFIGINFO;
```

The **VIOCONFIGINFO** structure contains configuration information about the screen.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

adapter Specifies the type of display adapter. It can be one of the following values:

Value	Meaning
DISPLAY_MONOCHROME	Monochrome/printer adapter
DISPLAY_CGA	Color graphics adapter
DISPLAY_EGA	Enhanced graphics adapter
DISPLAY_VGA	Video graphics array display adapter
DISPLAY_8514A	IBM Personal System/2 display adapter 8514/A

display Specifies the display/monitor type. It can be one of the following values:

Value	Meaning
MONITOR_MONOCHROME	Monochrome display
MONITOR_COLOR	Color display
MONITOR_ENHANCED	Enhanced color display
MONITOR_8503	8503 monochrome display
MONITOR_851X_COLOR	8512 or 8513 color display
MONITOR_8514	8514 color display

cbMemory Specifies the amount of memory in the adapter (in bytes).

Configuration Specifies the configuration ID requested. It can be one of the following values:

Value	Meaning
VIO_CONFIG_CURRENT	The current display adapter
VIO_CONFIG_PRIMARY	The primary display adapter
VIO_CONFIG_SECONDARY	The secondary display adapter

VDHVersion Reserved; must be zero.

Flags Specifies flag bits. The value 0x0001 sets default power-on configuration.

HWBufferSize Specifies the amount of memory required to save the full hardware state of the device adapter (not including the physical video buffer).

FullSaveSize Specifies the amount of memory required to save the entire physical video buffer.

PartSaveSize Specifies the amount of memory required to save the portion of the physical video buffer that will be overwritten by a pop-up window.

EMAdaptersOFF Specifies the offset to information that describes other display adapters emulated by this display adapter.

EMDisplaysOFF Specifies the offset to information that describes other display types emulated by this display.

See Also **VioGetConfig**

Changes

The following fields have been added to the end of the **VIOCONFIGINFO** structure:

```
USHORT  Configuration;
USHORT  VDHVersion;
USHORT  Flags;
ULONG   HWBufferSize;
ULONG   FullSaveSize;
ULONG   PartSaveSize;
USHORT  EMAdaptersOFF;
USHORT  EMDisplaysOFF;
```

VIOFONTCELLSIZE New

```
typedef struct _VIOFONTCELLSIZE {     /* viofcsz */
    LONG %cx%;
    LONG %cy%;
} VIOFONTCELLSIZE;
```

The **VIOFONTCELLSIZE** structure specifies the size of a font cell.

Fields

cx Specifies the width of the font cell.

cy Specifies the length of the font cell.

See Also

DevEscape

VIOMODEINFO Change

```
typedef struct _VIOMODEINFO {     /* viomi */
    USHORT cb;
    UCHAR  fbType;
    UCHAR  color;
    USHORT col;
    USHORT row;
    USHORT hres;
    USHORT vres;
    UCHAR  fmt_ID;
    UCHAR  attrib;
    ULONG  buf_addr;
    ULONG  buf_length;
    ULONG  full_length;
    ULONG  partial_length;
    PCH    ext_data_addr;
} VIOMODEINFO;
```

The **VIOMODEINFO** structure contains information about the screen mode.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

fbType Specifies the screen mode. It is one of the following values:

Value	Meaning
VGMT_OTHER	Set adapter to other than a monochrome/printer adapter. If this value is not given, the monochrome/printer adapter is assumed by default.

Value	Meaning
VGMT_GRAPHICS	Set graphics mode. If this value is not given, the adapter is set to text mode.
VGMT_DISABLEBURST	Disable color-burst mode. If this value is not given, color-burst mode is enabled.

color Specifies the number of colors (defined as a power of 2). This is equivalent to the number of color bits that define the color. It is one of the following values:

Value	Meaning
COLORS_2	2 colors
COLORS_4	4 colors
COLORS_16	16 colors

col Specifies the number of text columns.

row Specifies the number of text rows.

hres Specifies the number of pel columns (horizontal resolution).

vres Specifies the number of pel rows (vertical resolution).

fmt_ID Specifies the format of the attributes.

attrib Specifies the number of attributes in the **attribfmt** field.

buf_addr Specifies the 32-bit physical address of the physical video buffer for this mode.

buf_length Specifies the length (in bytes) of the physical video buffer for this mode.

full_length Specifies the size (in bytes) of the buffer required to save the entire physical video buffer for this mode.

partial_length Specifies the amount of memory required to save a portion of the physical video buffer for this mode. This portion of the physical video buffer is what is overwritten by a pop-up window.

ext_data_addr Specifies the far address of an extended-mode data structure, or zero if there is none.

See Also

VioGetMode, VioSetMode

Changes

The following fields have been added to the end of the **VIOMODEINFO** structure:

```
UCHAR  fmt_ID;
UCHAR  attrib;
ULONG  buf_addr;
ULONG  buf_length;
ULONG  full_length;
ULONG  partial_length;
PCH    ext_data_addr;
```

VIOSETTARGET New

```
typedef struct _VIOSETTARGET {     /* viosett */
    USHORT  cb;
    USHORT  type;
    USHORT  defaultalgorithm;
} VIOSETTARGET;
```

The **VIOSETTARGET** structure identifies the target display of the next call to the **VioSetMode** function.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

type Specifies the request type. To retrieve the target information, this field must be set to 0x0006.

defaultalgorithm Specifies the target display of the next call to the **VioSetMode** function. A value of 0x0000 specifies the default display (the active display when the computer was powered on), 0x0001 specifies the primary display, and 0x0002 specifies the secondary display.

See Also

VioGetState, VioSetMode, VioSetState

VIOSETULINELOC New

```
typedef struct _VIOSETULINELOC {     /* viouline */
    USHORT cb;
    USHORT type;
    USHORT scanline;
} VIOSETULINELOC;
```

The **VIOSETULINELOC** structure contains the location of the underline.

Fields

cb Specifies the size of the structure (in bytes). Programs written in the C language should use the **sizeof** operator to set this field.

type Specifies the request type. To retrieve the underline location, this field must be set to 0x0005.

scanline Specifies the location of the underline. This value is normally in the range 0 through - 1 (the value of the scan line minus 1). A value of 32 means that underlining is disabled.

See Also

VioGetState, VioSetState

■ VIOSIZECOUNT New

```
typedef struct _VIOSIZECOUNT {    /* viosz */
    LONG %MaxCount%;
    LONG %Count%;
} VIOSIZECOUNT;
```

The **VIOSIZECOUNT** structure contains the size of the **VIOFONTCELLSIZE** structure.

Fields

MaxCount Specifies the maximum number of occurrences of the **VIOFONTCELLSIZE** structure.

Count Specifies the actual number of occurrences of the **VIOFONTCELLSIZE** structure.

See Also

DevEscape

■ WNDPARAMS Change

```
typedef struct _WNDPARAMS {    /* wprm */
    USHORT fsStatus;
    USHORT cchText;
    PSZ    pszText;
    USHORT cbPresParams;
    PVOID  pPresParams;
    USHORT cbCtlData;
    PVOID  pCtlData;
} WNDPARAMS;
```

The **WNDPARAMS** structure contains information about a window.

Fields

fsStatus Specifies the window parameters which are to be set or queried. This can be any combination of the following values:

Value	Meaning
WPM_TEXT	Text
WPM_CTLDATA	Control data
WPM_PRESPARAMS	Presentation parameters
WPM_CCHTEXT	Size of text
WPM_CBCTLDATA	Size of control data
WPM_CBPRESPARAMS	Size of presentation parameters

cchText Specifies the length of the window text.

pszText Points to the window text.

cbPresParams Specifies the length of the presentation parameters.

pPresParams Points to the **PRESPARAMS** structure that contains presentation parameters. This field is NULL if there are no presentation parameters.

cbCtlData Specifies the length of the class-specific data.

pCtlData Points to the class-specific data.

See Also

PRESPARAMS, WM_QUERYWINDOWPARAMS, WM_SETWINDOWPARAMS

Changes

The following constants have been defined for the **fsStatus** field:

Value	Meaning
WPM_TEXT	Text
WPM_CTLDATA	Control data
WPM_PRESPARAMS	Presentation parameters
WPM_CCHTEXT	Size of text
WPM_CBCTLDATA	Size of control data
WPM_CBPRESPARAMS	Size of presentation parameters

Index

A
AVAILDATA, 363

B
Bit masks, 7

C
C-language format, 5
Calling conventions, 6
CBM_HILITE, 42, 57
CBM_ISLISTSHOWING, 42, 57
CBM_SHOWLIST, 42, 57
CBN_EFCHANGE, 41, 42, 58
CBN_EFCSCROLL, 42
CBN_EFSCROLL, 58
CBN_ENTER, 42, 58
CBN_LBSCROLL, 42, 59
CBN_LBSELECT, 41, 42
CBN_LBSELECT, 59
CBN_MEMERROR, 42, 59
CBN_SHOWLIST, 42, 60
CHARBUNDLE, 363
Combination-box control, 40–42
Combo box, 40–42
  messages sent from, 42
  messages sent to, 42
Constant names, 11

D
DENA1, 364
DevEscape, 60
DevPostDeviceModes, 66
DevQueryCaps, 68
DOS-compatibility mode, filenames, 20
DosAllocHuge, 72
DosAllocSeg, 74
DosAllocShrSeg, 76
DosChgFilePtr, 19
DosCopy, 77
DosCreateSem, 79
DosCreateThread, 80
DosDevIOCtl2, 81
DosEditName, 21, 82
DosEnterCritSec, 83
DosEnumAttribute, 84
DosExecPgm, 21
DosExitCritSec, 85
DosExitList, 86
DosFileIO, 88
DosFileLocks, 19
DosFindFirst, 21, 23
DosFindFirst2, 90
DosFindNext, 23, 93
DosFreeResource, 95
DosFreeSeg, 95
DosFSAttach, 19, 96
DosFSCtl, 18, 97
DosGetDBCSEv, 99
DosGetModHandle, 100
DosGetResource, 101
DosGetResource2, 102
DosGetVersion, 103
DosLoadModule, 104
DosMakePipe, 105
DosMkDir2, 23, 106
DosMonReg, 107
DosOpen, 19, 108
DosOpen2, 23, 113
DosQFHandState, 117
DosQFileInfo, 23, 119
DosQFSAttach, 121
DosQNmPipeInfo, 123
DosQNmPipeSemState, 124
DosQPathInfo, 23, 124
DosQSysInfo, 20
DosRead, 19, 126
DosReadAsync, 128
DosReadQueue, 130
DosReallocHuge, 131
DosReallocSeg, 133
DosSearchPath, 134
DosSemClear, 135
DosSemRequest, 136
DosSetFileInfo, 137
DosSetMaxFH, 139
DosSetPathInfo, 139
DosSetPrty, 141
DosSetVec, 142
DosShutdown, 143
DosStartSession, 144
DosSubAlloc, 145
DosSubFree, 146
DosWaitNmPipe, 146
DosWrite, 19, 147
DosWriteAsync, 148
DosWriteQueue, 150

E
EAOP, 365
EM_QUERYREADONLY, 152
EM_SETINSERTMODE, 152
EM_SETREADONLY, 152
EM_SETTEXTLIMIT, 153
EN_CHANGE, 153
EN_INSERTMODETOGGLE, 153
EN_KILLFOCUS, 154
EN_MEMERROR, 154
EN_OVERFLOW, 154
EN_SCROLL, 155
EN_SETFOCUS, 155
Entry field, multiple-line, 42–50
ENTRYFDATA, 365
Error, file-system, 21–22
Extended attribute, 23–28
  data type, 24
  data-type field, 27
  naming, 23

F
F1 key, 32, 35
FAT (file allocation table), 17–18
FATTRS, 366
FEA, 367
FEALIST, 368
Field names, 8
File allocation table (FAT), 17–18
File system *See* Installable file system
File-system error, 21–22
FILEFINDBUF2, 368
Filename
  conventions, 20
  DOS-compatibility mode, 20
  metacharacter, 21
FILESTATUS2, 369
FIOLOCKCMD, 370
FIOLOCKREC, 370
FIOREADWRITE, 371
FIOSEEKCMD, 371
FIOUNLOCKCMD, 372
FIOUNLOCKREC, 372
FONTMETRICS, 373
FSINFO, 377
FSQBUFFER, 378

G
GEA, 379
GEALIST, 379
GpiCallSegmentMatrix, 156
GpiCreateLogFont, 158
GpiDestroyPS, 160
GpiGetData, 160
GpiLoadFonts, 162
GpiOutlinePath, 163
GpiPlayMetaFile, 164
GpiPolyLine, 167
GpiQueryBitmapBits, 168
GpiQueryCharDirection, 170
GpiQueryCharStringPos, 171
GpiQueryDefArcParams, 173
GpiQueryDefAttrs, 173
GpiQueryDefTag, 175
GpiQueryDefViewingLimits, 176
GpiQueryFontFileDescriptions, 176
GpiQueryMetaFileBits, 177
GpiResetPS, 178
GpiRotate, 180
GpiScale, 181
GpiSetCharDirection, 182
GpiSetDefArcParams, 183
GpiSetDefAttrs, 184
GpiSetDefTag, 186
GpiSetDefViewingLimits, 186
GpiSetPS, 187
GpiSetViewingLimits, 189
GpiTranslate, 190
GpiUnloadFonts, 191
GpiWCBitBlt, 192

H
HCINFO, 380
Help
  button, 32, 37
  hook, 38
  instance, 33, 37
  library, 35
  menu, 32, 25
  table, 34
Help Manager, 31–40
  messages sent by, 39
  messages sent to, 39–40
HELPINIT, 33, 380
HELPTABLE, 34, 381
High-performance file system (HPFS), 17–18
HM_ACTIONBAR_COMMAND, 195
HM_CREATE_HELP_TABLE, 195
HM_DISMISS_WINDOW, 196
HM_DISPLAY_HELP, 37, 196
HM_ERROR, 32, 197
HM_EXT_HELP, 37, 198
HM_EXT_HELP_UNDEFINED, 199
HM_HELP_CONTENTS, 199
HM_HELP_INDEX, 37, 199
HM_HELPSUBITEM_NOT_FOUND, 200
HM_INFORM, 201
HM_KEYS_HELP, 37, 201
HM_LOAD_HELP_TABLE, 202
HM_QUERY_KEYS_HELP, 202
HM_REPLACE_HELP_FOR_HELP, 202
HM_SET_ACTIVE_WINDOW, 34, 203
HM_SET_HELP_LIBRARY_NAME, 203
HM_SET_HELP_WINDOW_TITLE, 204
HM_SET_SHOW_PANEL_ID, 204
HM_TUTORIAL, 205
HPFS (high-performance file system), 17–18

I
Include files, 5
Information Presentation Facility Compiler (IPFC), 32
Initialization file, 28–31
Installable file system, 17–23
IPFC (Information Presentation Facility Compiler), 32

K
KbdCharIn, 205
KbdGetHWID, 207
KBDHWID, 382
KBDKEYINFO, 383
KbdRegister, 208

L
LDTADDRINFO, 384
LINFOSEG, 384
LM_INSERTITEM, 41
Local file system, 19

M
MATRIXLF, 386
MLE_SEARCHDATA, 387
MLECTLDATA, 388
MLEFORMATRECT, 390
MLEMARGSTRUCT, 390
MLEOVERFLOW, 391
MLM_CHARFROMLINE, 48, 210
MLM_CLEAR, 43, 45, 48, 211
MLM_COPY, 45, 48, 211
MLM_CUT, 45, 48, 212
MLM_DELETE, 44, 48, 212
MLM_DISABLEREFRESH, 45, 48, 212
MLM_ENABLEREFRESH, 45, 48, 213
MLM_EXPORT, 44, 48, 213
MLM_FORMAT, 45, 48, 214
MLM_IMPORT, 44, 48, 214
MLM_INSERT, 43, 48, 215
MLM_LINEFROMCHAR, 48, 215
MLM_PASTE, 45, 48, 216
MLM_QUERYBACKCOLOR, 44, 48, 216
MLM_QUERYCHANGED, 46, 48, 217
MLM_QUERYFIRSTCHAR, 46, 48, 217
MLM_QUERYFONT, 44, 48, 218
MLM_QUERYFORMATLINELENGTH, 45, 48, 218
MLM_QUERYFORMATRECT, 44, 48, 219
MLM_QUERYFORMATTEXTLENGTH, 45, 48, 220
MLM_QUERYIMPORTEXPORT, 48, 221
MLM_QUERYLINECOUNT, 44, 48, 221
MLM_QUERYLINELENGTH, 48, 222
MLM_QUERYREADONLY, 44, 48, 222
MLM_QUERYSEL, 43, 49, 222
MLM_QUERYSELTEXT, 45, 49, 223
MLM_QUERYTABSTOP, 44, 49, 224
MLM_QUERYTEXTCOLOR, 44, 49, 224
MLM_QUERYTEXTLENGTH, 44, 49, 225
MLM_QUERYTEXTLIMIT, 44, 49, 225
MLM_QUERYUNDO, 43, 49, 226
MLM_QUERYWRAP, 44, 49, 226
MLM_RESETUNDO, 43, 49, 227
MLM_SEARCH, 46, 49, 227

MLM_SETBACKCOLOR, 44, 49, 228
MLM_SETCHANGED, 46, 49, 229
MLM_SETFIRSTCHAR, 46, 49, 229
MLM_SETFONT, 44, 49, 230
MLM_SETFORMATRECT, 44, 49, 230
MLM_SETIMPORTEXPORT, 44, 49, 232
MLM_SETREADONLY, 44, 49, 232
MLM_SETSEL, 43, 49, 233
MLM_SETTABSTOP, 44, 49, 233
MLM_SETTEXTCOLOR, 44, 49, 234
MLM_SETTEXTLIMIT, 44, 49, 234
MLM_SETWRAP, 49, 235
MLM_UNDO, 43, 49, 235
MLN_CHANGE, 46, 49, 236
MLN_CLPBDFAIL, 49, 236
MLN_HSCROLL, 46, 50, 237
MLN_KILLFOCUS, 50, 237
MLN_MARGIN, 50, 238
MLN_MEMERROR, 50, 238
MLN_OVERFLOW, 50, 238
MLN_PIXHORZOVERFLOW, 50, 239
MLN_PIXVERTOVERFLOW, 50, 240
MLN_SEARCHPAUSE, 50, 240
MLN_SETFOCUS, 50, 241
MLN_TEXTOVERFLOW, 50, 241
MLN_UNDOOVERFLOW, 50, 241
MLN_VSCROLL, 46, 50, 242
MM_DISMISSMENU, 242
MM_QUERYSELITEMID, 242
MOU_DISPLAYMODECHANGE, 243
MOU_SETPROTDRAWADDRESS, 243
MOU_SETREALDRAWADDRESS, 244
MOU_UPDATEDISPLAYMODE, 245
MOU_VER, 246
MouGetNumQueEl, 247
MouSynch, 248
Multiple-line entry field (MLE), 42–50
  copying text, 45
  editing text, 43
  exporting text, 44
  formatting text, 44
Multiple-line entry field (MLE) *(continued)*
  importing text, 44
  messages sent from, 49
  messages sent to, 48
  notification code, 46
  pasting text, 45
  searching and replacing text, 46

N
Naming conventions, 8–10
Notational conventions, 11

O
Overviews, 13–50

P
PARAM, 392
Parameter names, 8
PicIchg, 249
PicPrint, 250
PIPESEMSTATE, 393
PL_ALTERED, 250
Prefixes, 9
PRESPARAMS, 393
PrfAddProgram, 29, 30, 250
PrfChangeProgram, 252
PrfCloseProfile, 29, 253
PrfCreateGroup, 29, 30, 253
PrfDestroyGroup, 254
PrfOpenProfile, 29, 255
PRFPROFILE, 394
PrfQueryDefinition, 255
PrfQueryProfile, 30, 257
PrfQueryProfileData, 29, 258
PrfQueryProfileInt, 30, 259
PrfQueryProfileSize, 29, 260
PrfQueryProfileString, 29, 30, 261
PrfQueryProgramCategory, 262
PrfQueryProgramHandle, 263
PrfQueryProgramTitles, 264
PrfRemoveProgram, 265
PrfReset, 30, 265
PrfWriteProfileData, 266
PrfWriteProfileString, 29, 267
Profile Manager, 28–31
PROGDETAILS, 394
PROGTITLE, 395
PROGTYPE, 396
Pseudo-character device, 19
PTRACEBUF, 397
PTRDRAWDATA, 400

R
Remote file system, 19

S
SBCDATA, 400
SBM_SETTHUMBSIZE, 268
SCR_ALLOCLDT, 269
SCR_ALLOCLDTOFF, 269
SCR_DEALLOCLDT, 270
Single-file device, 19
STATUSDATA, 401
Structures, 8, 361, 363–410
SWBLOCK, 401

T
TBM_TRACKMOVE, 271
TRACKINFO, 402
Types, 8–10, 361, 362

V
VIOCOLORREG, 404
VIOCONFIGINFO, 404
VioCreatePS, 272
VIOFONTCELLSIZE, 406
VioGetBuf, 273
VioGetConfig, 274
VioGetMode, 275
VioGetState, 276
VIOMODEINFO, 406
VioReadCellStr, 277
VioScrollDn, 278
VioScrollLf, 280
VioScrollRt, 281
VioScrollUp, 282
VioSetCurType, 283
VioSetMode, 284
VioSetState, 286
VIOSETTARGET, 408
VIOSETULINELOC, 408
VioShowBuf, 287
VIOSIZECOUNT, 409
VioWrtCellStr, 288
VioWrtNCell, 289

W
WinAddProgram, 290
WinAssociateHelpInstance, 33, 38, 291
WinBroadcastMsg, 292
WinCreateFrameControls, 293
WinCreateGroup, 293

WinCreateHelpInstance, 33, 294
WinCreateHelpTable, 295
WinCreatePointerIndirect, 296
WinCreateSwitchEntry, 296
WinCreateWindow, 41, 42, 47, 298
WinDefDlgProc, 33
WinDefWindowProc, 33
WinDeleteLibrary, 300
WinDeleteProcedure, 300
WinDestroyHelpInstance, 38, 300
WinDlgBox, 37
WinDrawBitmap, 301
WinEnumDlgItem, 302
WinGetDlgMsg, 303
WinGetLastError, 38
WinGetNextWindow, 304
WinGetSysBitmap, 304
WinInitialize, 29, 33
WinInstStartApp, 306
WinIsWindowShowing, 307
WinLoadHelpTable, 308
WinLoadLibrary, 308
WinLoadProcedure, 309
WinLockWindow, 309
WinQueryActiveWindow, 309
WinQueryAnchorBlock, 310
WinQueryCapture, 310
WinQueryClipbrdOwner, 310
WinQueryClipbrdViewer, 311
WinQueryDefinition, 311
WinQueryFocus, 312
WinQueryHelpInstance, 33, 313
WinQueryPresParam, 313
WinQueryProfileData, 315
WinQueryProfileInt, 316
WinQueryProfileSize, 317
WinQueryProfileString, 318
WinQueryProgramTitles, 318
WinQuerySessionTitle, 320
WinQuerySwitchEntry, 320
WinQuerySwitchHandle, 321
WinQuerySwitchList, 322
WinQuerySysModalWindow, 322
WinQuerySysValue, 323
WinQueryTaskSizePos, 326
WinQueryWindow, 327
WinQueryWindowLockCount, 328
WinRegisterClass, 328
WinReleasePS, 330
WinRemovePresParam, 331
WinSetPresParam, 332
WinSetSysColors, 333
WinSetSysValue, 336
WinSetWindowPos, 339
WinSwitchToProgram, 342
WinTerminateApp, 343
WinWindowFromID, 343
WinWindowFromPoint, 344
WinWriteProfileData, 345
WinWriteProfileString, 346
WM_ADJUSTWINDOWPOS, 347
WM_APPTERMINATENOTIFY, 348
WM_CALCFRAMERECT, 348
WM_CALCVALIDRECTS, 349
WM_CHAR, 350
WM_CLOSE, 352
WM_COMMAND, 36
WM_CONTROL, 42
WM_DRAWITEM, 352
WM_FORMATFRAME, 353
WM_MEASUREITEM, 354
WM_MOVE, 355
WM_PRESPARAMCHANGED, 355
WM_QUERYHELPINFO, 356
WM_SAVEAPPLICATION, 357
WM_SETHELPINFO, 357
WM_WINDOWPOSCHANGED, 358
WNDPARAMS, 409

Invest in CD-ROM Technology!

Microsoft® Programmer's Library is the ultimate programmer's reference on a single CD-ROM disc. It contains full text of the OS/2 Software Development Kit (SDK) manuals, the Windows SDK manuals, most Microsoft Language manuals, and several Microsoft Press books written for the serious programmer, including THE MS-DOS ENCYCLOPEDIA. Plus 20 floppies' worth of "clip art" sample code. Navigate through this mass of programming knowledge with boolean searches and hypertextual links between related data. The price is $395 suggested retail— a fraction of the price for this material in print form.

For a limited time, Programmer's Library is available System Complete with the Denon DRD-253 CD-ROM drive for $949. That's a savings of $575!

And if you order now, you'll receive FREE Microsoft Audio Software ($99 value) that turns your CD-ROM drive into a programmable CD audio player. So you can *listen* to a keyboard for a change.

Call (800) 227-4679 for details.

For a Free Demo Disk* please

Print your name and address:

NAME

COMPANY NAME (if applicable)

STREET ADDRESS

CITY STATE ZIP

DAYTIME TELEPHONE (in case we have questions about your order)

Check the appropriate box

☐ **1.2 MB floppy Demo Disk:** Contains the self-running demo as well as self-guided and interactive demos of the features of Programmer's Library. Also includes a portion of the actual Programmer's Library database.

☐ **1.44 MB floppy Demo Disk:** Same as above for 3.5 inch drives.

☐ **360K floppy Demo Disk:** The self-running demo showing the impressive features of Programmer's Library.

Send this coupon to:

Microsoft Corporation ▪ Attn: Special Promotions, Dept 127
16011 NE 36th Way ▪ Box 97017 ▪ Redmond, WA 98073-9717

*Offer Valid While Supplies Last

MICROSOFT® OS/2 Programmer's Reference
Including Presentation Manager

The MICROSOFT OS/2 PROGRAMMER'S REFERENCE LIBRARY is a low-cost way to explore Presentation Manager's Application Programming Interface (API) and start creating intuitive, easy-to-use software applications for its graphical environment. The library of four volumes—companions to the affordably priced Microsoft OS/2 Presentation Manager Softset development tools—is packed with detailed information on every MS® OS/2 system function, related data types, macros, structures, messages, and file formats.

Each volume in the series is written by a team of OS/2 specialists—many involved in the development and ongoing enhancement of OS/2 at Microsoft. The MICROSOFT OS/2 PROGRAMMER'S REFERENCE LIBRARY is the cornerstone of every OS/2 developer's programming library.

Volume 1

Volume 1 details the conceptual framework of the MS OS/2 Application Programming Interface (API). Included are thorough descriptions of MS OS/2 programming models, overviews of basic programming considerations, and explanations of the interaction between the API and the rest of the MS OS/2 system. Sections include *Introducing MS OS/2; Window Manager; Graphics Programming Interface;* and *System Services.*

Volume 2

Volume 2 is a comprehensive, alphabetic listing of MS OS/2 Presentation Manager version 1.1 functions and of the structures used with these functions. Each function entry includes information on syntax; descriptions of the function's actions and purpose; parameters and field definitions; return values, error values, and restrictions; source-code examples; and programming notes. A separate chapter discusses file formats. Appendixes cover error values and device capabilities.

Volume 3

Similar in format to Volume 2, Volume 3 is a comprehensive, alphabetic listing of MS OS/2 version 1.1 base functions, including their structures and file formats. Appendixes cover error values and ANSI escape sequences.

Volume 4

Volume 4 contains information on the new API elements in MS OS/2 version 1.2, as well as updated information on version 1.1 functions, structures, and messages.

U.S.A.	**$19.95**
U.K.	£18.95
Austral.	$29.95

(recommended)

ISBN 1-55615-259-0

51995

9 781556 152597

Computers/Operating Systems/OS/2